**MASTER EN SCIENCES INFORMATIQUES**

# Étude et implémentation de primitives de synchronisation de processus coopérants

De Boeck, Pierre

*Award date:*
1987

*Awarding institution:*
Universite de Namur

Facultés Universitaires Notre-Dame de la Paix
Institut d'Informatique
Année académique 1986-1987

# *Etude et implémentation de primitives de synchronisation de processus coopérants*

DE BOECK Pierre

Mémoire présenté en vue de l'obtention
du grade de Licencié et Maître en informatique

# REMERCIEMENTS

Je désire remercier ici les personnes qui m'ont aidé dans la résolution de ce mémoire, et plus particulièrement :

- Monsieur Henri Leroy, directeur de ce mémoire et professeur à l'Institut d'Informatique des F.U.N.D.P. pour sa disponibilité et ses précieux conseils,
- Monsieur Baudoin Lecharlier, professeur à l'Institut d'Informatique des F.U.N.D.P. pour le temps important qu'il a consacré à la lecture et à la critique de mon travail,
- Monsieur Axel Van Lansweerde, professeur à l'Institut d'Informatique des F.U.N.D.P. et Monsieur Pierre de Marneffe, professeur à l'Université de Liège, pour leurs nombreuses critiques interessantes sur les primitives étudiées dans ce travail.

Deux primitives de synchronisation de processus coopérants, le *sémaphore* et le *monitor*, sont étudiées dans ce mémoire. Cette étude aboutit à la proposition d'une nouvelle primitive, le *monitor concurrent*, ayant à la fois les avantages du *monitor* (synchronisation élégante) et ceux du *sémaphore* (performances interessantes). L'implémentation de cette primitive ainsi que les problèmes posés par celle-ci sont ensuite étudiés.

# STRUCTURE DU MEMOIRE

- Introduction.

- Chapitre 1 : problèmes de synchronisation posés par la programmation concurrente.

- Chapitre 2 : le sémaphore.

- Chapitre 3 : le monitor.

- Chapitre 4 : le monitor concurrent.

- Chapitre 5 : implémentation.

- Conclusion.

- Annexes.

- Bibliographie.

# INTRODUCTION

Depuis quelques années, on observe une tendance de plus en plus grande à concevoir des systèmes informatiques composés de multiples processeurs, communiquant entre eux au moyen de mémoires communes et / ou par le biais de réseaux de communication. La baisse constante du coût des processeurs et les gains de performances possibles sont deux raisons parmi d'autres qui ont favorisé l'apparition de ces nouvelles architectures.

Un programme exécuté sur de tels systèmes n'est plus vu comme une série d'instructions traitées séquentiellement mais comme un ensemble de tâches ou processus exécutés simultanément et communiquant entre eux en accédant à des ressources communes. De tels programmes sont appelés programmes concurrents et les langages de programmation permettant de les définir sont appelés langages de programmation concurrente. Ces langages doivent offrir au programmeur les "outils" nécessaires à la résolution des problèmes posés par ce nouveau type de programmation. Ces problèmes sont par exemple la définition des processus, l'activation et la terminaison des processus, leur synchronisation,...

Etant donné l'importance que peut avoir la synchronisation des processus sur des priorités telles que la correction et l'efficacité d'un programme, il est indispensable qu'un langage de programmation concurrente de haut-niveau offre au programmeur les outils qui lui permettent de réaliser le plus aisément possible une synchronisation correcte et efficace. De tels outils sont appelés communément "primitives de synchronisation".

Les primitives de synchronisation sont donc un domaine qui fait l'objet d'intenses recherches depuis quelques années, ce qui explique le nombre important de différentes formes de primitives qui ont été proposées. Ce mémoire s'interesse à l'une d'entre elles, le *monitor*, qui est certainement une des primitives les plus célèbres et les plus largement utilisées dans les langages de programmation concurrente actuels. Cette primitive fut proposée au début des années 70 par HOARE [9] et HANSEN [16] , et permet, à la différence d'autres primitives plus simples comme le *sémaphore* [2] de réaliser facilement une synchronisation élégante (abstraction des ressources et de la synchronisation des opérations accédant aux

ressources, contrôle de qui accède aux ressources et de ce que l'on fait sur ces ressources,...).

Le premier objectif de ce travail sera de montrer, par la résolution d'exemples concrets les avantages du monitor par rapport au sémaphore, mais aussi ses limites, qui se situent principalement au niveau des performances. Nous constaterons en effet que la résolution de certains problèmes où un haut degré de concurrence est possible peut entraîner de graves baisses de performances si l'on utilise le monitor à la place du sémaphore.

Le second objectif sera alors de trouver une primitive qui permette à la fois de conserver les avantages du monitor mais aussi d'atteindre les mêmes performances que celles que l'on peut obtenir avec le sémaphore. Nous verrons que cette primitive sera obtenue par quelques modifications (syntaxiques et sémantiques) du monitor.

Enfin, le dernier objectif sera de trouver une implémentation correcte et efficace de cette nouvelle primitive (que l'on appelera le *monitor concurrent*).

Pour réaliser ces objectifs, nous avons structuré ce mémoire de la manière suivante :

Un premier chapitre présentera la programmation concurrente et les problèmes de synchronisation posés par ce type de programmation. Ce chapitre introduira, au moyen de nombreux exemples, les notions de - processus séquentiel
- exécution concurrente (ou simultanée) de processus séquentiels
- synchronisation de processus coopérants, c-à-d de processus séquentiels exécutés en concurrence et qui ont accès à un ensemble de ressources communes.
- primitives de synchronisation.

Les deux chapitres suivants présenteront respectivement le sémaphore et le monitor. Pour chacune de ces primitives, nous verrons - son principe, sa syntaxe, sa sémantique
- la résolution, avec cette primitive, de problèmes concrets
- ses avantages et ses inconvénients.

Nous verrons notamment que le principal avantage du monitor réside dans les possibilités qu'il offre pour réaliser une synchronisation élégante et son principal inconvénient se situe au niveau des performances. Pour le sémaphore, c'est l'inverse : synchronisation

peu élégante mais performances interessantes.

Le chapitre quatre présentera le *monitor concurrent*, qui est un monitor qui a été modifié afin d'avoir à la fois les avantages du monitor (élégance) et du sémaphore (performances).

Enfin, le dernier chapitre présentera les implémentations du monitor concurrent et du monitor conventionnel (le monitor original). Pour chacune de ces implémentations, nous décrirons un ensemble de règles permettant leur simplification, ceci en vue d'améliorer leur efficacité. De plus, nous essaierons de montrer que ces implémentations réalisent une synchronisation correcte (nous aurons bien sûr défini auparavant le terme "synchronisation correcte"). Nous terminerons ce chapitre par 2 problèmes que posent l'implémentation d'un monitor concurrent.

Nous clotûrerons ce mémoire par un conclusion générale sur celui-ci : nous rappelerons les objectifs que l'on s'était fixé, la façon dont ces objectifs ont été réalisés et les points qui peuvent encore faire l'objet de futures recherches.

# CHAPITRE UN

# problèmes de synchronisation posés par la programmation concurrente

## 1.1. Processus séquentiel

## 1.2. Exécution concurrente de processus séquentiels

1.2.1. Indétermination relative à la concurrence
1.2.2. processus asynchrones
1.2.3. Processus synchrones (coopérants)

## 1.3. Synchronisation de processus "coopérants"

1.3.1. Problèmes de synchronisation
    1.3.1.1. Exclusion mutuelle
    1.3.1.2. Condition de synchronisation
1.3.2. Exemples
    1.3.2.1. Exemple 1
    1.3.2.2. Exemple 2
    1.3.2.3. Exemple 3
1.3.3. Primitives de synchronisation

Un programme concurrent décrit un ensemble de processus séquentiels qui seront exécutés simultanément. Avant d'examiner les problèmes posés par ce type d'exécution, nous allons préciser les notions de processus séquentiel et de concurrence.

## 1.1. PROCESSUS SEQUENTIEL.

Un processus séquentiel consiste en un ensemble de données "locales" au processus et une liste d'opérations sur ces données, opérations qui seront exécutées strictement les unes après les autres dans l'ordre spécifié par la liste.

Nous représenterons un processus par la notation habituelle qui consiste à décrire les données du processus et la liste d'opérations exécutées par ce processus. Cette notation est utilisée par exemple dans [1], [2], [3], [4], [5].

```
process < process name >
var     < description des données locales du processus >;
begin   < liste d'opérations > end ;
```

*Remarques* : 1. Le mot "donnée" doit être pris au sens large : cela peut être des variables simples, des tableaux, des listes, des fichiers,...

2. Les données d'un processus ne peuvent être référencées que par les opérations de celui-ci.

3. La description des données et/ou des opérations se fera au moyen d'une notation de type **PASCAL** [6].
(Une opération étant une suite d'instructions **PASCAL**)

*Exemple 1.*

```
process  echange ;
  var    a,b : integer ;
  begin  read(a) ; read(b) ;
         a := a+b ;
         b := a-b ;
         a := a-b
  end ;
```

L'exécution du processus *échange* a pour effet d'échanger les valeurs initiales de *a* et *b*. Les valeurs initiales sont données au moyen des opérations *read(a)* et *read(b)*.

*Exemple 2.*

```
process  minimum ;
  var    a ; array [1. .n] of integer ;
         min , i : integer ;
  begin  i := 1 ; min := a[1] ;
         while i < n
            do begin i := i + 1 ;
                     if   min > a[i]
                     then min := a[i]
               end
  end ;
```

L'exécution du processus *minimum* a pour effet d'affecter à *min* la valeur minimum du vecteur $a[1. .n]$, $n \geq 1$.

*Exemple 3* : supposons qu'une firme vende ses produits par le biais d'un ensemble de représentants (par exemple 3 / province, c-à-d 27 représentants). A la fin du mois, la firme dispose d'un fichier *commande* par représentant (au total, il y a donc 27 fichiers *commande*). Le fichier *commande* du représentant n° *i* (désigné par *commande(i)*) contient l'ensemble des commandes passées par ce représentant au cours du mois (un enregistrement = une commande).

La firme souhaite créer pour chaque représentant un fichier *commande-bis*. Le fichier *commande-bis* du représentant n° *i* (désigné par *commande-bis(i)*) contiendrait l'ensemble des commandes passées par ce représentant au cours du mois, pour lesquelles le montant est $\geq$ 2000 Fb (on suppose qu'à chaque commande est associée un champ *montant* indiquant le montant de la commande. Si *x* indique une commande, *x.montant* désignera son champ *montant*).

On peut donc dire que le fichier *commande-bis(i)* comprend l'ensemble des enregistrements de *commande(i)* dont le champ *montant* contient une valeur $\geq$ 2000.

$$commande\text{-}bis(i) = \left\{ \begin{array}{l} x \text{ tq } - \; x \in commande(i) \\ \qquad - \; x.montant \geq 2000 \end{array} \right\}$$

Implémentation : nous allons créer pour chaque représentant n° *i* un processus de nom *rep(i)*, dont l'exécution aura pour effet de créer le fichier *commande-bis(i)* à partir de *commande(i)*.

Chaque processus *rep(i)* aura comme données un fichier *commande* et un fichier *commande-bis* (initialement vide).

```
process  rep(i) ;        {1 ≤ i ≤ 27}
  var    commande, commande-bis : file of com ;
         x : com ;
  begin  open(commande) ;
         open(commande-bis) ;
         while        not (eof(commande))
            do begin  read(commande,x) ;
                      if x.montant ⩾ 2000
                      then write(commande-bis,x)
               end ;
         close(commande-bis) ;
         close(commande)
  end ;
```

*Remarques* : 1. Le type *com* spécifie le format d'un enregistrement *commande*. Un enregistrement de type *com* contient un champ *montant*, indiquant la valeur de la commande référencée par l'enregistrement. Le type *com* est accessible à tous les processus *rep(i)*, $1 \leq i \leq 27$.

2. *On suppose qu'il existe une clause permettant d'établir* un "lien" entre une variable de type fichier (*":file of ...;"*) et le fichier réel. Par exemple, les variables *commande* et *commande-bis* déclarées dans *rep(i)* seront liées respectivement aux fichiers *commande(i)* et *commande-bis(i)*.

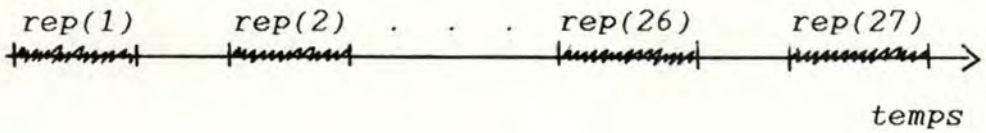
3. Afin d'assurer la localité des données d'un processus (les données d'un processus ne sont accessibles que par ce processus), on imposera que des variables de type fichier appartenant à des processus distincts soient liées à des fichiers distincts.

4. Les commandes *open*, *close*, *read* et *write* sont les opérations habituelles sur un fichier. Elles ont respectivement pour effet d'ouvrir un fichier et de pointer vers le premier enregistrement de celui-ci, de fermer un fichier, de lire l'enregistrement pointé* sur le suivant, d'écrire à la fin du fichier. La commande *eof* permet de détecter la fin du fichier.

Dans le 3° exemple, nous avons 27 processus séquentiels, chacun produisant son fichier *commande-bis* respectif. Le souhait de la firme (avoir un fichier *commande-bis* par représentant) sera réalisé lorsque les 27 processus auront été exécutés.

Une première façon d'obtenir ces 27 fichiers est d'exécuter les processus l'un après l'autre. Par exemple, "on" active l'exécution de *rep(1)*. Lorsque *rep(1)* est terminé, on active *rep(2)*, et ainsi de suite ("on" ici peut être un opérateur, un processus,...).

On peut représenter ce type d'exécution par un graphique du type :




où *rep(i)* représente l'intervalle de temps durant lequel *rep(i)* est exécuté. Dans ce type d'exécution, les intervalles de temps des processus seront toujours disjoints deux à deux.

Ce type d'exécution présente des inconvénients à deux niveaux [7] : - d'abord au niveau des spécifications : l'exécution strictement ordonnée des processus a pour effet de surspécifier le problème. En effet, ce type d'exécution impose

1) une production strictement ordonnée des fichiers *commande-bis*.

2) *un ordre précis dans la production de ces fichiers.*

Or, les spécifications originales demandaient simplement que les 27 fichiers *commande-bis* soient produits, sans préciser la façon dont ils seront produits.

- ensuite au niveau des performances : il est évident que ce type d'exécution "maximise" le temps qu'il faut pour produire les 27 fichiers *commande-bis*.

* et de pointer

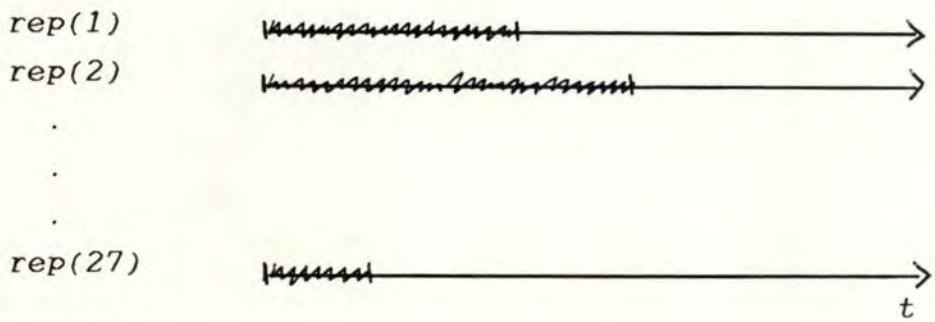
Nous avons vu qu'une première façon d'exécuter un ensemble de processus séquentiels était de les exécuter strictement les uns après les autres.

L'autre façon est une exécution concurrente, c-à-d simultanée des processus.

## 1.2. EXECUTION CONCURRENTE DE PROCESSUS SEQUENTIELS.

Des processus sont exécutés en concurrence si leur exécution n'est pas strictement ordonnée. Si on représente une exécution concurrente au moyen d'un graphique représentant les intervalles de temps d'exécution (cfr p I.4), on constate que ceux-ci ont des segments en commun.

*Exemple* : reprenons l'exemple 3 et exécutons en concurrence les 27 processus ("on" active les 27 processus en même temps). Représentons leur exécution au moyen d'un graphique :




*Remarque* : Une exécution concurrente de processus peut être réalisée de différentes façons [7] :

*a) time-sharing (*)* : plusieurs processus partagent un **CPU**, et à intervalles réguliers, un *process scheduler* retire le **CPU** au processus actif et l'alloue à un autre processus (multiplexage).

*b) multiprocessing* : plusieurs processus partagent plusieurs **CPU**s. Ces **CPU**s peuvent accéder à une mémoire commune.

N.B. : la multiprogrammation est une forme particulière de multiprocessing où les processus se partagent deux types de **CPU** : des **CPU**s pour les opérations ordinaires et des **CPU**s pour les opérations d'*entrée/sortie*.

(*) Si dans la mesure du possible, nous éviterons les termes anglais, il en existe cependant qui sont intraduisibles en français!

c) *Distributed processing* : plusieurs processus partagent plusieurs CPU. Chaque CPU a sa propre mémoire, et communique avec les autres par le biais d'un réseau de communication.

N.B. : Un langage de programmation concurrente de haut-niveau doit permettre une programmation qui soit la plus indépendante possible de la façon dont est réalisée l'exécution concurrente.

### *1.2.1. INDETERMINATION RELATIVE A LA CONCURRENCE*

Une des caractéristiques importantes de l'exécution concurrente est son comportement "*non-déterministe*". Lorsqu'on exécute un processus séquentiel, on peut déterminer avant l'exécution, l'ordre dans lequel seront exécutées les opérations du processus. En effet, il suffit de connaître la liste de ses opérations et l'état initial de ses données. On dit alors qu'un processus séquentiel a un comportement déterministe.

Par contre, lorsqu'un ensemble de processus sont exécutés en concurrence, il est impossible de déterminer à l'avance l'ordre dans lequel seront exécutées des opérations appartenant à des processus distincts. Cet ordre dépend en effet de la "vitesse" à laquelle est exécuté chaque processus, vitesse dépendant elle-même de caractéristiques qui sont ignorées dans un contexte de programmation de haut-niveau : par exemple, des caractéristiques relatives aux processeurs (nbre d'instructions / sec, fréquence d'interruptions externes,...), à l'operating-system (time-slice, organisation des files d'attente,...),... La seule hypothèse que le programmeur peut faire est que la "vitesse" de chaque processus est > 0, c-à-d qu'un processus ne sera jamais privé du processeur indéfiniment [8].

*Exemple* :

```
process  P ;                    process  Q ;
  var    x,y : integer ;          var    i,j : integer ;
  begin  x := x+y ;               begin  i := i*j ;
         write(x)                        write(i)
  end ;                           end ;
```

Lorsque *P* et *Q* sont exécutés en concurrence, on peut déterminer à l'avance l'ordre dans lequel seront exécutées les opérations de *P* (*x := x+y* puis *write(x)*) et de *Q* (*i := i*j* puis *write(i)*).

Par contre, on ne peut pas déterminer l'ordre dans lequel seront exécutées une opération de *P* et une opération de *Q*. Par exemple, *x := x+y* peut être exécutée avant *i := i*j*, ou bien après ou encore en même temps.

Selon que, pour un ensemble de processus concurrents, on accepte toutes les formes d'exécution, ou que l'on en refuse certaines, on parle de processus *asynchrones* ou *synchrones*.

### *1.2.2. PROCESSUS ASYNCHRONES.*

On a vu que lorsqu'on exécutait un ensemble de processus concurrents, il existait plusieurs formes possibles d'exécution entre deux opérations *op1* et *op2* appartenant à des processus différents.

soit *op1* avant *op2*
*op2* avant *op1*
*op1* et *op2* en concurrence.

Lorsque pour tout couple d'opérations appartenant à des processus distincts, on (le programmeur) accepte toutes formes d'exécution, on dit que les processus sont asynchrones : chaque processus s'exécute sans tenir compte de l'état d'exécution des autres processus.

*Exemple* : si on reprend l'exemple 3 (*cfr p I.2*), on a 27 processus asynchrones : chaque processus produit son fichier *commande-bis* sans se préoccuper de savoir où en sont les autres.

### *1.2.3. PROCESSUS SYNCHRONES (COOPERANTS).*

Lorsque pour certains couples d'opérations appartenant à des processus distincts, on impose certaines conditions sur la séquence d'exécution, on dit que les processus sont synchrones.

Par exemple, on impose que telle opération d'un processus soit toujours exécutée avant telle opération d'un autre processus. Ou bien que telle opération d'un processus soit toujours exécutée avant ou après telle opération d'un autre processus, mais jamais en même temps.

On pourrait se demander pourquoi "synchroniser" certains processus ? En fait, jusqu'à présent, nous avons vu uniquement des exemples de processus indépendants, ç-à-d qui n'accèdent qu'à leurs propres données. Ces processus n'ont donc aucune *ressource* en commun et l'exécution de chacun d'eux se déroule en totale indépendance (exemple des 27 processus où chacun d'eux possède ses fichiers *commande* et *commande-bis*).

Cette forme d'indépendance, intéressante par sa simplicité, a cependant ses limites. En effet, des processus ont généralement accès à un ensemble de ressources communes, cela pour différentes raisons. Cela peut être par exemple pour des raisons d'"économie" : des processus ont accès à la même imprimante, au même disque,...

Des processus peuvent aussi avoir besoin de communiquer (échange d'informations, de signaux,...) : cette communication se fait de nouveau par un partage de ressources telles que des variables, des lignes de communication,...

Le reste de ce chapitre sera consacré aux problèmes de synchronisation posés par ce partage de ressources (*). Nous nous limiterons à des ressources de type variable simple, tableaux, liste, fichier,...c-à-d les types de ressources habituellement déclarés dans des langages de haut-niveau. Les problèmes de synchronisation rencontrés seront cependant les mêmes pour d'autres types de ressources, habituellement invisibles du programmeur.

N.B. : l'ensemble des ressources communes sera déclaré à "l'extérieur" des processus. Les opérations d'un processus pourront donc faire référence aux données locales de ce processus et / ou aux ressources communes.

(*) Dorénavant, nous utiliserons le terme *processus coopérants* pour désigner un ensemble de processus concurrents ayant accès à un ensemble de ressources communes.

## 1.3. SYNCHRONISATION DE PROCESSUS "COOPERANTS".

Cette section présente d'abord les deux types de synchronisation qui sont nécessaires lorsque des processus coopérent, c-à-d ont accès à des ressources communes. Une série d'exemples seront ensuite présentés, chaque exemple étant une version plus ou moins complexe du "problème des représentants" (*cfr p I.2*). Enfin, le concept de primitive de synchronisation sera présenté, permettant l'introduction des chapitres suivants.

### *1.3.1. PROBLEMES DE SYNCHRONISATION.*

Les spécialistes de la programmation concurrente comme *DIJKSTRA* [2], *HANSEN* [8] [11], *HOARE* [9], *HABERMAN* [10], s'accordent pour reconnaître deux grands types de synchronisation : l'exclusion mutuelle et la condition de synchronisation.

#### 1.3.1.1. Exclusion mutuelle.

Imposer une contrainte d'exclusion mutuelle sur un ensemble d'opérations consiste à interdire toute exécution concurrente de ces opérations. Par exemple, imposer une exclusion mutuelle entre *op1* et *op2* consiste à interdire toutes exécutions concurrentes de *op1* et *op2* : *op1* peut être exécutée avant ou après *op2*, mais jamais en même temps.

*Quel est le but de l'exclusion mutuelle ?*

L'exclusion mutuelle vise à garantir qu'une opération donnera des résultats prévisibles et cohérents. L'incohérence des résultats d'une opération est due à des interférences survenues pendant son exécution. Par interférences, nous entendons le fait que des variables référencées par une opération soient modifiées "à son insu" c-à-d soient modifiées au cours de son exécution par d'autres opérations (des interférences entre deux opérations ne peuvent donc avoir lieu que si ces opérations accèdent à une même ressource, et une au moins de ces opérations modifie l'état de la ressource).

Afin d'éviter ces interférences, il suffit donc d'exclure mutuellement tout couple d'opérations dont l'une peut modifier des variables référencées par l'autre.

*Exemple* : soit l'opération **op** : $buffer[0] := x$ ;
$buffer[1] := -x$ ;

Si au moment où commence l'exécution de cette opération, $x = x_0$, le résultat cohérent à la fin de l'exécution est le suivant :

$$\{ x = x_0 , buffer[0] = x_0 , buffer[1] = -x_0 \}$$

Supposons cependant que $buffer[0..1]$ soit une variable commune. Il se peut alors qu'au cours de l'exécution de **op**, d'autres opérations produisent des interférences, en modifiant $buffer[0]$ et / ou $buffer[1]$. Par exemple, entre le moment où "$buffer[0] := x$" se termine et celui où "$buffer[1] := -x$" commence, un autre processus exécute une opération contenant l'instruction "$buffer[0] := 0$".
Dans ce cas, **op** se termine avec le résultat suivant :

$$\{ x = x_0 , buffer[0] = 0 , buffer[1] = -x_0 \}$$

*Ce résultat n'est évidemment pas du tout celui qu'on attendait.* Pour que **op** donne toujours un résultat cohérent, il suffit d'exclure mutuellement **op** avec toutes opérations qui modifient $buffer[0]$ et / ou $buffer[1]$ (on suppose que $x$ est une variable locale au processus qui exécute *op*).

### 1.3.1.2. Condition de synchronisation.

Une condition de synchronisation est une expression booléenne, portant sur les variables communes, associée à une opération. Une opération ne pourra être exécutée que lorsque la condition de synchronisation qui lui est associée est vraie.

Prenons par exemple un processus *P1* contenant une opération *op*, à laquelle est associée une condition de synchronisation *B*.
Lorsque *P1* voudra exécuter *op*, deux cas peuvent se présenter :

soit *B* est vrai et *op* est alors exécutéé.

soit *B* est faux et l'exécution de *op* est alors suspendue jusqu'à ce que d'autres processus rendent la condition *B* vraie.

*A quoi sert la condition de synchronisation ?*

Supposons que l'on ait l'opération "*s := s-1*" où *s* est une variable commune.

Afin de garantir des résultats cohérents, l'opération "*s := s-1*" doit être mutuellement exclusive avec toute opération pouvant modifier *s*. Peu importe le contexte où cette opération est utilisée, la règle d'exclusion mutuelle sera toujours la même :
"*s := s-1*" doit être mutuellement exclusive avec toute opération pouvant modifier *s*.

Le problème de la condition de synchronisation est lui complètement différent, et dépend entièrement du contexte dans lequel sera exécutée l'opération.

Par exemple, si l'on utilise "*s := s-1*" dans un contexte où la valeur de *s* doit être toujours $\geq 0$, la condition de synchronisation de l'opération sera la suivante : $s > 0$ (en supposant que la valeur initiale de *s* est $\geq 0$, cela permet de maintenir la propriété "$s \geq 0$", à tout instant).

Si par contre, "*s :=s-1*" est utilisée dans un contexte où la valeur de *s* peut être quelconque, l'opération ne sera soumise à aucune condition de synchronisation.

Le problème de l'exclusion mutuelle et de la condition de synchronisation sont donc deux problèmes totalement différents : le premier vise à garantir le déroulement correct d'une opération en empêchant toutes les interférences qui pourraient survenir.

Le deuxième vise à ordonnancer un ensemble d'opérations afin de maintenir certaines propriétés sur les variables communes, propriétés choisies par le programmeur en fonction du problème à résoudre.

### *1.3.2. EXEMPLES.*

Nous allons présenter trois exemples permettant de bien comprendre les concepts d'exclusion mutuelle et de condition de synchronisation.

1.3.2.1. Exemple 1.

On reprend les spécifications du "problème des représentants" (*cfr p I.2*) avec les modifications suivantes : la firme souhaite avoir en fin de mois un seul fichier *commande-2000* contenant l'ensemble des commandes dont le montant est $\geqslant$ 2000 (il n'y a donc plus de fichier *commande-bis.*)

$$commande\text{-}2000 = \left\{ x \text{ tq } -\exists\, 1 \leqslant i \leqslant 27 \text{ tq } x \in commande(i) \right\}$$
$$- x.montant \geqslant 2000$$

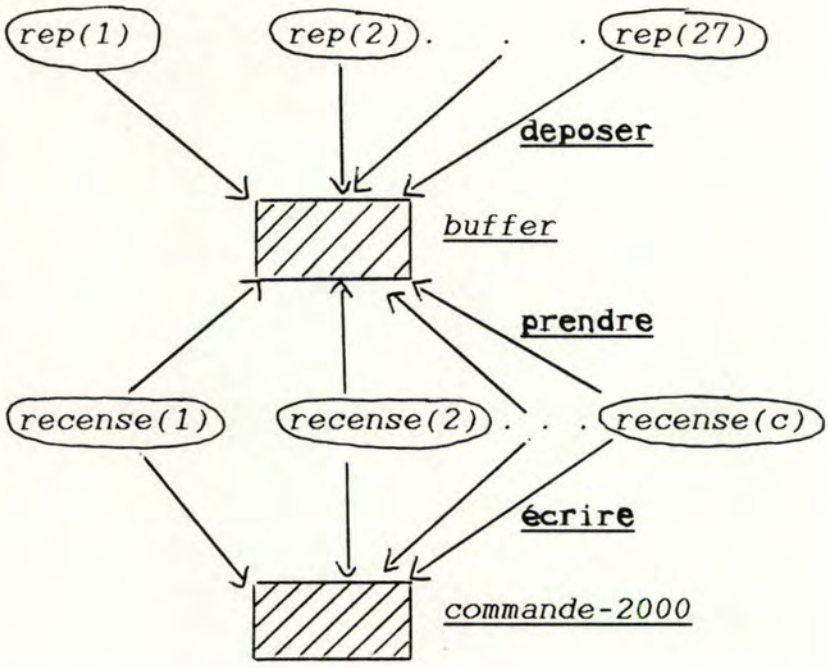
*Comment implémenter cela ?*

Nous allons - garder les 27 processus *rep(i)*
- créer *c* processus *recense(j)* $(c \geqslant 1)$

Le travail de chaque processus *rep(i)* consiste à lire son fichier *commande(i)*, et à passer chaque commande ayant un montant $\geqslant$ 2000 à un des processus *recense(j)*.

Le travail de chaque processus *recense(j)* consiste à recevoir les commandes des processus *rep(i)*, et à écrire ces commandes sur le fichier *commande-2000*.

Le shéma ci-dessous présente l'ensemble des ressources nécessaires à cette implémentation, et les opérations sur ces ressources :




Il y a deux ressources communes principales.

*a) Le buffer* : il est déclaré comme une variable de type *commande* :

*buffer : com ;*

Ce buffer est de taille 1 : il ne peut contenir qu'une seule commande à la fois.

L'objectif du buffer est de permettre le passage des commandes entre les processus *rep(i)* et les processus *recense(j)*. Les processus *rep(i) viennent* **déposer** *une commande dans le buffer et les* processus *recense(j)* viennent **prendre** une commande du buffer.

Un ordonnancement de ces opérations est nécessaire, afin d'éviter qu'une commande soit "écrasée" ou "dupliquée" :

- une opération **prendre** ne peut être exécutée que lorsque le buffer contient "une commande qui n'a pas encore été prise".
  Ceci permet d'éviter qu'une commande soit dupliquée, c-à-d prise deux fois.
- une opération **deposer** ne peut être exécutée que lorsque le buffer ne contient pas de commande qui n'a pas encore été prise.
  Ceci permet d'éviter qu'une commande soit écrasée.

N.B. : initialement, le buffer est vide : la première opération qui pourra donc être exécutée est une opération **deposer**.

L'implémentation de cet ordonnancement se réalise avec les conditions de synchronisation. On va associer au buffer une variable *s* de type booleen, avec la signification suivante :

*s* = *true* $\Leftrightarrow$ le buffer contient une commande qui n'a pas encore été prise.

Initialement : *s* = *false* (car le buffer est vide).

Les opérations **deposer** seront alors associées à la condition de synchronisation suivante : *s* = *false*.

Les opérations **prendre** seront alors associées à la condition de synchronisation suivante : *s* = *true*.

Lorsqu'un processus *rep(i)* veut deposer une commande *x* dans le buffer, il utilisera l'opération **deposer** qui a la forme suivante :

$\{ s = false \}$ (*)
*buffer* := *x* ;
*s* := *true* ;

Lorsqu'un processus *recense(j)* veut prendre une commande du buffer et la mettre dans *x*, il utilisera l'opération **prendre** qui a la forme suivante :

$\{ s = true \}$
*x* := *buffer* ;
*s* := *false* ;

(*) Une opération précédée de $\{B\}$ signifie que l'opération est suspendue tant que *B* est faux et lorsque *B* est vrai, l'opération est exécutée (*B* est une expression booléenne).

*Contraintes d'exclusion mutuelle* :

- les opérations **deposer** doivent être exécutées strictement les unes après les autres.
- les opérations **prendre** doivent être exécutées strictement les unes après les autres.
- Qu'en est-il entre une opération **deposer** et une opération **prendre** ? Avec l'ordonnancement imposé (une opération **deposer** est exécutée lorsque *s = false* ; une opération **prendre** est exécutée lorsque *s = true*), ces deux opérations ne seront jamais exécutées en concurrence : il est donc inutile d'imposer une exclusion mutuelle.

Nous allons maintenant voir la deuxième ressource :

*b) commande-2000* : Cette ressource est un fichier commun aux processus *recense(j)*, et est déclarée de la façon suivante :

*commande-2000 : file of com ;*

Un processus *recense(j)* déposera une commande *y* dans ce fichier au moyen de l'opération **écrire**, qui a le code suivant :

*write (commande-2000 , y) ;*

Les opérations **ecrire** ne seront soumises à aucun ordonnancement particulier (pour ce problème-ci !). Par contre, elles devront être exécutées strictement l'une après l'autre.

On voit bien ici que le problème de la condition de synchronisation est fort dépendant du contexte dans lequel les opérations sont utilisées. On aurait pu par exemple utiliser l'opération **ecrire** dans un contexte où l'on impose que deux opérations **ecrire** successives soient exécutées par deux processus *recense(j)* distincts. Dans ce cas, l'opération **ecrire** aurait été soumise à une condition de synchronisation du type : "le processus qui a exécuté l'opération **ecrire** précédente est distinct du processus qui veut exécuter l'opération **ecrire** actuelle.

*Code des processus.*

Nous allons montrer le code d'un processus *rep(i)* et d'un processus *recense(j)*. Le buffer et le fichier *commande-2000 seront déclarés à l'extérieur des processus, de la manière suivante :*

```
var buffer : com ; s : boolean ; commande-2000 : file of com ;
```

*a) Code d'un processus rep(i).*

```
process rep(i) ;
  var   commande : file of com ;
        x : com ;

  begin open(commande) ;
        while not (eof(commande))
        do    begin read(commande,x) ;
                    if x.montant ⩾ 2000
                    then {s = false}
                          buffer := x ;
                          s := true          ] deposer
              end ;
        close(commande)

  end ;
```

N.B. : rappelons que l'opération **deposer** sera exécutée en exclusion mutuelle avec toutes opération **deposer**.

*b) Code d'un processus recense(j).*

```
process recense(j) ;
  var   y : com ;

  begin while true
        do    begin {s = true}
                    y := buffer ;          ]  prendre
                    s := false ;           ]
                    write(commande-2000,y) ]  ecrire
              end

  end ;
```

*Remarques* : 1. Nous ne nous sommes pas préoccupés des problèmes d'initialisation (activation des processus, initialisation des ressources communes,...) et de cloture (fermeture de *commande-2000*,...).

2. Nous ne nous sommes pas préoccupés de l'implémentation des contraintes de synchronisation (exclusion mutuelle et / ou condition de synchronisation) pour la bonne raison que nous ne disposons pas encore de *primitives de synchronisation.*

3. *On aura remarqué que le problème présenté ici est en fait simplement le "problème des producteurs-consommateurs"* [9], [10], [11], où les processus *rep(i)* sont les producteurs et les processus *recense(j)* sont les consommateurs.

1.3.2.2. Exemple 2.

Le deuxième exemple est identique au premier, excepté la taille du buffer, qui pourra contenir jusqu'à $N$ commandes $(N \geq 1)$.

*buffer : array [0..N-1] of com ;*

Comme dans le premier exemple, un processus *rep(i)* pourra mettre une commande dans le buffer au moyen d'une opération **deposer**. De même, un processus *recense(j)* pourra prendre une commande du buffer au moyen de l'opération **prendre**.

Les commandes seront enlevées du buffer dans leur ordre d'arrivée (*FIFO*), cela afin d'éviter qu'une commande soit "oubliée" indéfiniment.

N.B. : en fait, cette règle (FIFO) serait réellement indispensable uniquement si le nombre de commandes était infini, ce qui n'est pas le cas ici.

*Comment implémenter les opérations* "**prendre**" *et* "**déposer**" *?*

Nous allons d'abord décrire les trois états possibles dans lesquels une cellule du buffer peut se trouver :

*a) cellule vide* : une cellule se trouve dans cet état lorsque
- .) aucune opération (**deposer** ou **prendre**) n'est en cours sur cette cellule.
- .) la cellule ne contient pas de commande qui n'a pas encore été prise.

*b) cellule pleine* : une cellule se trouve dans cet état lorsque
- .) aucune opération n'est en cours sur cette cellule.
- .) la cellule contient une commande qui n'a pas encore été prise.

*c) cellule instable* : cellule qui n'est ni vide ni pleine (une opération y est en cours).

- Une opération **deposer** devra :
  - . trouver une cellule vide.
  - . deposer la commande dans cette cellule.

  Cette opération fera passer une cellule de l'état vide à l'état plein, via l'état instable :

  VIDE ⟶ INSTABLE ⟶ PLEIN

- Une opération **prendre** devra :
  - . trouver une cellule pleine (la plus "vieille").
  - . prendre la commande de cette cellule.

  Cette opération fera passer la cellule de l'état plein à l'état vide, via l'état instable :

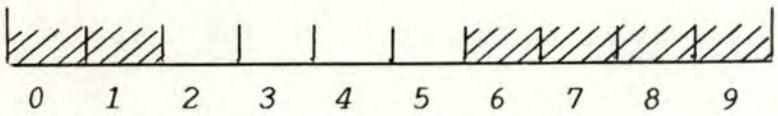
  PLEIN ⟶ INSTABLE ⟶ VIDE

Une façon d'implémenter cela est d'imposer que toutes les cellules pleines forment un bloc continu. Ainsi, lorsque aucune opération **deposer** ou **prendre** n'est en cours, le buffer contient deux blocs continus :

- un bloc *plein* contenant les cellules *pleines* dans leur ordre de remplissage.
- un bloc *vide* contenant les cellules *vides*.

Ces deux blocs sont contigus : le début de chaque bloc est toujours adjacent à la fin de l'autre bloc.

*Exemple* :




bloc *plein* : cellules *6, 7, 8, 9, 0, 1.*
bloc *vide* : cellules *2, 3, 4, 5.*

Ce type d'implémentation est appelé *buffer en anneau* $[2]$, $[10]$, car le buffer est perçu comme un anneau : la dernière cellule du buffer est adjacente à la première.

L'intêret de cette implémentation est sa simplicité : deux informations seulement sont nécessaires pour connaître l'adresse de toutes les cellules pleines, dans leur ordre de remplissage et l'adresse de toutes les cellules vides. En effet, il suffit de connaître : - l'adresse de la première cellule du bloc *plein (i).*
- l'adresse de la première cellule du bloc *vide (j).*

L'intervalle $[\, i,j \,[$ contient alors les cellules pleines et $[\, j,i \,[$ contient les cellules vides. Dans l'exemple ci-dessus, $i = 6$ et $j = 2$.

Nous allons donc déclarer deux variables *i* et *j*, de type entier permettant d'obtenir ces deux adresses.

La variable *i* indiquera le nombre d'opérations **prendre** qui ont été exécutées ; la variable *j* indiquera le nombre d'opérations **deposer** qui ont été exécutées. On peut affirmer que *(j-i)* indiquera donc le nombre de commandes qui reste dans le buffer, c-à-d la longueur du bloc *plein*.

*Quelle sera l'adresse de la première cellule du bloc plein ?*

Etant donné que la $k^{\circ}$ opération (**prendre** ou **deposer**) se fera sur la $(k-1) \bmod N$ cellule, on peut affirmer que les $(j-i)$ commandes dans le buffer se trouvent dans les cellules $(i \bmod N)$ jusque $((j-1) \bmod N)$. (Cela évidemment si $j-i > 0$).

De même, les $N - (j-i)$ emplacements libres se trouvent dans les cellules $(j \bmod N)$ jusque $((i-1) \bmod N)$. (Cela si $N - (j-i) > 0$).

*Nous pouvons maintenant donner le code des opérations* **deposer** *et* **prendre**.

a) ***deposer*** : cette opération va enlever la première cellule du bloc *vide*, et la mettre à la fin du bloc *plein*. Cette opération ne pourra donc être exécutée que lorsque la longueur du bloc *vide* est > 0 et la longueur du bloc *plein* est < *N*, ce qui revient à définir la condition de synchronisation suivante : $j-i < N$. Cela revient à dire que la différence entre le nombre d'opérations **deposer** et **prendre** ne peut jamais dépasser *N*.

$\{ j-i < N \}$
*buffer* [ *j mod N* ] := *x* ; ] **deposer**
*j* := *j*+1 ;

(*x* est une variable locale à un processus *rep(i)* et contient une commande).

b) ***prendre*** : cette opération va enlever la première cellule du bloc *plein*, et l'ajouter à la fin du bloc *vide*. Cette opération est donc soumise à la condition de synchronisation suivante : $j-i > 0$. Cela revient à dire que le nombre d'opérations **prendre** ne peut jamais dépasser le nombre d'opérations **deposer**.

$\{ j-i > 0 \}$
*x* := *buffer* [ *i mod N* ] ; ] **prendre**
*i* := *i*+1 ;

c) *Contraintes d'exclusion mutuelle* :

- les opérations **prendre** doivent être mutuellement exclusives.
- les opérations **deposer** doivent être mutuellement exclusives.
- entre une opération **deposer** et une opération **prendre**, il n'y aura pas d'interférences (*) car selon l'ordonnancement imposé, *i mod N* ne sera jamais égal à *j mod N* *lorsque les deux opérations sont en cours, et de plus, l'opération* **deposer** *ne peut pas rendre fausse la condition de synchronisation* de **prendre**, et vice-versa.
  $\Longrightarrow$pas d'exclusion mutuelle.

*Déclaration des ressources communes.*

```
var  buffer : array [ 0..N-1 ] of com ;
     i,j : integer ;   { initialement : i := 0 ; j := 0 }
     commande-2000 : file of com ;
```

Nous ne redonnerons pas le code d'un processus *rep(i)* ou *recense(j)* car il est exactement le même que celui décrit à la page *I.17-18*, excepté le code des opérations **prendre** et **deposer**.

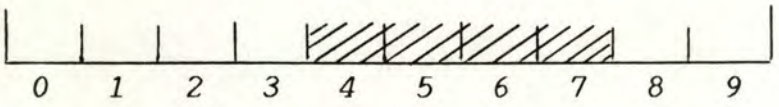
### 1.3.2.3. Exemple 3.

Dans l'exemple précédent, nous avons montré une implémentation d'un buffer, connue sous le nom de "buffer en anneau" : le buffer contient deux blocs contigus, l'un formé de cellules vides et l'autre de cellules pleines (les cellules de chaque bloc étant adjacentes). Il suffit alors de deux informations pour connaître la location de chaque cellule (pleine / vide).

Cette implémentation, avantageuse par sa simplicité, est cependant peu efficace : les contraintes de CONTINUITE dans les blocs (adjacence des cellules) ne permettent pas l'exécution concurrente de plusieurs opérations **deposer** (**prendre**) alors qu'il y a peut-être plusieurs cellules VIDES (PLEINES).

(*) Le fait qu'une opération puisse rendre fausse la condition de synchronisation d'une autre opération doit bien sûr être considéré aussi comme une interférence.

*Exemple* :




*bloc plein : 4, 5, 6, 7.*
*bloc vide : 8, 9, 0, 1, 2, 3.*

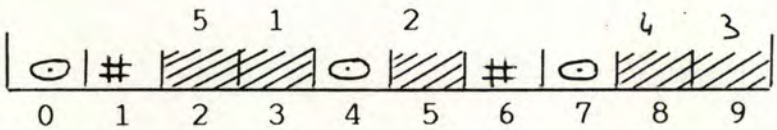
Supposons que trois opérations **deposer** demandent à être exécutées : afin d'être sûr que le bloc *plein* restera CONTINU, et que ses cellules seront "rangées" dans leur ordre de remplissage, il est nécessaire que l'opération **deposer** sur la cellule *9* ne soit exécutée que lorsque l'opération sur la cellule *8* est terminée. Les opérations **deposer** doivent donc être exécutées strictement l'une après l'autre.

Nous nous proposons dans ce 3° exemple de présenter une autre implémentation, permettant l'exécution concurrente d'opérations **deposer** et / ou **prendre** sur le buffer.

*Implémentation à base de listes.*

L'exécution concurrente d'opérations **deposer** et / ou **prendre** a comme conséquences de faire disparaître la propriété de "contiguïté" : les cellules pleines (vides) ne sont plus du tout adjacentes et sont dispersées dans le buffer.

*Exemples* : à un instant donné, l'état du buffer pourrait être le suivant




: *la i° cellule pleine.*

 : *cellule vide.*

 : *cellule instable.*

- Les cellules *1*, *6* sont instables (des opérations y sont en cours).
- Les cellules *3*, *5*, *9*, *8*, *2* sont les cellules pleines, dans leur ordre de remplissage.
- Les cellules *0*, *4*, *7* sont les cellules vides.

On voit que le bloc des cellules pleines (vides) n'est plus du tout continu. Il est donc nécessaire de disposer de deux listes d'*indices* permettant de connaître :

- l'adresse des cellules pleines dans leur ordre de remplissage (ce sera la liste PL).
- l'adresse des cellules vides (liste VD).

*N.B.* : - tous les éléments d'une liste sont distincts.
- $PL \cap VD = 0$

*Une opération* **deposer** *se décompose alors en trois sous-opérations :*

*d1* : trouver l'indice d'une cellule vide
⇒prendre un élément quelconque de la liste *VD* (soit *v* cet élément). A cet instant, *buffer*[*v*] est instable.

*d2* : deposer la commande *x* dans la cellule vide
*buffer*[*v*] := *x* ;

*d3* : *buffer*[*v*] devient plein ⇒ ajouter *v* en fin de la liste *PL* (parmi les cellules pleines, *buffer*[*v*] est la dernière)

Condition de synchronisation de **deposer** (ou plutôt de *d1*) :
"la liste *VD* doit être non-vide".

*Une opération* **prendre** *se décompose aussi en trois sous-opérations :*

*p1* : trouver l'indice de la plus vieille cellule pleine.
⇒ prendre le premier élément de la liste *PL* (soit *p* cet élément). A cet instant, *buffer*[*p*] est instable.

*p2* : prendre la commande de la cellule pleine et la mettre dans *x*.
*x* := *buffer*[*p*] ;

*p3* : *buffer*[*p*] devient vide ⇒ ajouter *p* dans la liste *VD*, à un endroit quelconque.

Condition de synchronisation de **prendre** (ou plutôt de *p1*) :
"la liste *PL* doit être non-vide".

*Contraintes d'exclusion mutuelle* :

.) *l'opération d1* : cette opération devra être mutuellement exclusive avec toute opération modifiant la liste *VD*.
$\Longrightarrow$ *d1* sera mutuellement exclusive avec des opérations *d1* ou *p3*.

.) *l'opération d2* : *(buffer[v] := x)*
Aucune opération ne peut interférer avec *d2*.
- c'est trivial pour les opérations *d1*, *d3*, *p1*, *p2*, *p3*.
- deux opérations *d2* ont chacunes un *v* distinct $\Longrightarrow$pas d'interférences.

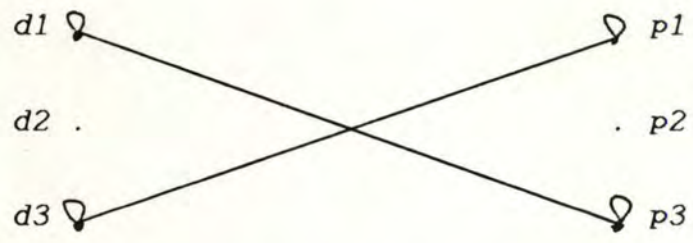
$\Longrightarrow$ *d2* ne doit être mutuellement exclusive avec aucune opération.

.) *l'opération d3* : cette opération devra être mutuellement exclusive avec toute opération modifiant la liste *PL*.
$\Longrightarrow$ *d3* sera mutuellement exclusive avec des opérations *d3* ou *p1*.

.) *les opérations p1, p2, p3* : on applique le même raisonnement.

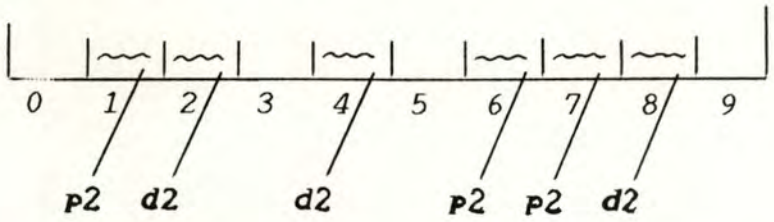
On peut représenter ces contraintes au moyen d'un graphe non orienté :

$G(x,u)$ où $X = \{ d1, d2, d3, p1, p2, p3 \}$
$U = \{ (x_1, x_2)$ tq - $x_1, x_2 \in X$
- $x_1$ est mutuellement exclusif avec $x_2$ $\}$




Cette implémentation est beaucoup plus efficace car à tout instant, plusieurs opérations *d2* peuvent être en cours, chacune déposant une commande dans une cellule vide et plusieurs opérations *p2* peuvent être en cours, chacunes prenant une commande d'une cellule vide.

*Exemple* :




*6 opérations* **p2** ou **d2** sont en cours $\Longrightarrow$ les cellules *1, 2, 4, 6, 7,* et *8* sont instables. Les autres cellules sont soit dans *PL*, soit dans *VD*.

*Implémentation des listes PL et VD.*

Il reste à montrer comment implémenter les listes *PL* et *VD*, et donner le code des opérations qui les manipulent (*d1, d3, p1, p3*).

Les listes *PL* et *VD* sont en fait des listes de nombres entiers compris entre 0 et N-1 (rappelons que les deux listes sont disjointes et contiennent chacune des nombres distincts). Une manière simple de représenter de telles listes est d'associer au buffer, un tableau *T* de type entier et de taille *N* :

*T : array [0..N-1] of integer ;*

Une liste sera alors représentée au moyen d'une variable *I* et du tableau *T* de la manière suivante ($0 \leq I \leq N$) :

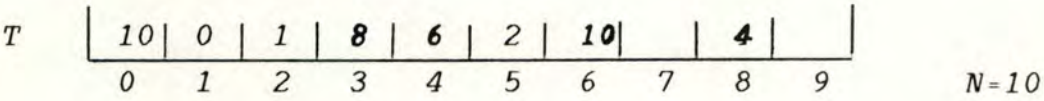
la liste (*I,T*) est vide si $I=N$

la liste (*I,T*) = { valeur de *I*, liste (*T(I),T*) } si $I<N$

*Exemple* : on veut représenter - la liste { *3, 8, 4, 6* }

- la liste { *5, 2, 1, 0* }

On va déclarer deux variables *I* et *J* : $I=3$ *et* $J=5$




(*I,T*) représente la liste { *3, 8, 4, 6* }

(*J,T*) représente la liste { *5, 2, 1, 0* }

Nous allons donc représenter les listes *PL* et *VD* au moyen :

- d'un tableau *T : array [0..N-1] of integer ;*
- de deux variables entières *PL, VD : integer ;*

De plus, nous associerons à chaque liste, une variable contenant le dernier élément de la liste $\Longrightarrow$ *fPL, fVD : integer ;* Ces deux variables n'ont de sens que lorsque leur liste respective est non-vide!

Initialement : - la liste *Pl* est vide $\Rightarrow$ *PL* := *N* ;
- la liste *VD* = $\{ 0, 1, 2, \ldots, N-1 \}$

$\Rightarrow$ *VD* := 0 ;
*T*[0] := 1 ;
*T*[1] := 2 ;
.
.
.
*T*[*N*-1] := *N* ;
*fVD* := *N*-1

*Code de l'opération d1* : mettre dans *v* un élément quelconque de *VD* (par exemple le premier élément).

$\Rightarrow \{ VD \neq N \}$ (condition de synchronisation de ***d1***)
*v* := *VD* ;
*VD* := *T*[*VD*] ; (on supprime le 1° élément de *VD*)

*Code de l'opération d3* : ajouter *v* à la fin de *PL*.

$\Rightarrow$ *if* *PL* ≠ *N* (PL ≠ vide)
*then* *T*[*fPL*] := *v* ; (ajouter *v* en fin de *PL*)
*T*[*v*] := *N* ;
*fPL* := *v*
*else* *PL* := *v* ; (crée une liste $\{ v \}$)
*T*[*v*] := *N* ;
*fPL* := *v* ;

L'opération **deposer** exécutée par un processus *rep(i)* a alors le code suivant :

$\{ VD \neq N \}$
**d1** [ *v* := *VD* ;
*VD* := *T*[*VD*] ;
**d2** [ *buffer*[*v*] := *x* ;
[ *if* *PL* ≠ *N*
*then* *T*[*fPL*] := *V* *else* *PL* := *v* ;
**d3** *T*[*v*] := *N* ;
*fPL* := *v* ;

N.B. : - *x* et *v* sont locales en processus *rep(i)*.
- les opérations *p1*, *p3* sont similaires à *d1*, *d3* excepté que *.VD* est remplacé par *PL* (et vice-versa).
*.fPL* est remplacé par *fVD*.
- l'ensemble des ressources communes est constitué du *buffer*, de *T*, de *PL*, *fPL*, *VD*, *fVD* et du fichier *commande-2000*.

### *1.3.3. PRIMITIVES DE SYNCHRONISATION.*

Nous avons vu que l'exécution d'une opération sur des ressources communes était soumise à deux types de conditions (éventuellement vides) : - la condition de synchronisation, permettant de respecter un certain ordonnancement des opérations.
- la condition d'exclusion mutuelle, visant à garantir le déroulement correct d'une opération.

Une opération ne pourra donc être exécutée que lorsque ses deux conditions sont vraies. Si ce n'est pas le cas, l'opération est suspendue jusqu'à ce que les conditions deviennent vraies.

Cependant, la suspension d'une opération **op** n'est pas nécessairement terminée lorsque ses deux conditions deviennent vraies. En effet, il se peut qu'à ce moment-là, une autre opération soit exécutée et rende les conditions fausses, prolongeant ainsi la suspension de **op**. Une opération risque donc d'être suspendue indéfiniment alors que ses conditions sont vraies un nombre infini de fois.

Par exemple, supposons qu'une opération **ecrire** de nom *op* (décrite en I.16) soit suspendue, car une autre opération **ecrire** est en cours. Lorsque celle-ci se termine, *op* pourrait être exécutée : cependant, il se peut qu'une autre opération **ecrire** soit exécutée avant, prolongeant ainsi la suspension de *op*. *op* risque ainsi de rester bloqué indéfiniment bien que ses conditions sont vraies un nombre infini de fois. (N.B. : lorsqu'on parle de "suspension d'une opération", il faut évidemment comprendre "suspension du processus voulant exécuter l'opération").

Nous dirons donc que des opérations sont correctement synchronisées lorsque : 1. les contraintes de synchronisation sont bien respectées, c-à-d qu'une opération n'est exécutée que lorsque ses deux conditions sont vraies.

2. Il n'y a pas de suspension inutile, c-à-d q'une opération n'est suspendue que lorsque ses deux conditions sont fausses.
3. Une opération n'est pas systématiquement oubliée, c-à-d qu'une opération sera suspendue indéfiniment uniquement si il existe un moment où ses deux conditions resteront fausses à jamais.

Un langage de programmation concurrente devra mettre à la disposition du programmeur un *ensemble de moyens* qui lui permettront de synchroniser correctement un ensemble d'opérations. Cet ensemble de moyens est désigné sous le nom de *primitives de synchronisation.*

On trouvera dans [*12*] une synthèse des principales primitives de synchronisation qui ont été proposées.

Les deux chapitres suivants présentent le *sémaphore* et le *monitor,* qui sont deux des primitives les plus célèbres : le monitor fut proposé afin de pallier aux inconvénients du sémaphore, inconvénients dus à son caractère "peu structuré" (*), ce qui en fait un outil peu adéquat dans un langage de programmation concurrente de haut-niveau.

Bien que l'objet principal de ce mémoire est le monitor, nous présentons le sémaphore pour deux raisons. D'une part, pour bien montrer les avantages du monitor sur des primitives de "bas-niveau", et d'autre part pour de simples raisons pratiques : le sémaphore sera notre primitive de base dans nos implémentations.

(*) Par primitive peu structurée, nous entendons le fait qu'elle permet difficilement une programmation structurée.

# CHAPITRE DEUX

# Le sémaphore

## 2.1. Principe du sémaphore.

## 2.2. Résolution de problèmes de synchronisation au moyen de sémaphores.

### 2.2.1. Résolution de problèmes généraux.

### 2.2.2. Résolution d'exemples concrets.

## 2.3. Inconvénients du sémaphore.

Le sémaphore est, parmi les primitives de synchronisation qui ont été proposées, une des primitives les plus célèbres et les plus anciennes. Elle fut proposée pour la première fois par *DIJKSTRA* [2], [13] dans le but de synchroniser des processus "systèmes". Un des langages de programmation concurrente les plus célèbres implémentant cette primitive est l'*ALGOL 68* [14].

Après avoir présenté le principe du sémaphore (mécanisme, syntaxe, sémantique, ...), nous montrerons au moyen d'exemples concrets comment synchroniser correctement (*) des processus au moyen de cette primitive. Nous terminerons ce chapitre par une présentation des inconvénients du sémaphore, ou en tout cas des principaux, inconvénients dus surtout à son caractère "peu structuré".

## 2.1. PRINCIPE DU SEMAPHORE.

Un sémaphore est une variable commune, de type entier et ne pouvant prendre que des valeurs $\geq 0$. Deux opérations sont définies sur un sémaphore :

- l'opération **V** : si $s$ est un sémaphore, **V**($s$) a pour effet d'incrémenter de 1 la valeur de $s$.
- l'opération **P** : si $s$ est un sémaphore, **P**($s$) a pour effet de décrémenter de 1 la valeur de $s$, dès que $s > 0$. (Tant que $s=0$, l'opération **P**($s$) est suspendue).
- de plus, une opération ***Init***($s,n$) permet d'initialiser le sémaphore $s$, en lui affectant la valeur $n$, avec $n \geq 0$.

Il est interessant de voir les contraintes de synchronisation auxquelles sont soumises les opérations **P** et **V** :

- l'opération **V** :
  - . condition d'exclusion mutuelle est qu'aucune opération **P** et / ou **V** ne soit en cours.
  - . il n'y a pas de condition de synchronisation.
- l'opération **P** :
  - . condition d'exclusion mutuelle est qu'aucune opération **P** et / ou **V** ne soit en cours.
  - . condition de synchronisation est $\{s>0\}$.

(*) Nous avons défini en *I.28-29* les critères auxquels devait répondre une synchronisation correcte d'opérations.

Nous supposerons que la synchronisation correcte des opérations **P** et **V** est "acquise d'avance", c-à-d qu'elle n'est pas à charge du programmeur. Le programmeur est donc assuré que :

1.a : une opération **V**(*s*) ne sera exécutée que lorsqu'aucunes opérations **P**(*s*) et / ou **V**(*s*) ne sont en cours.
b : une opération **P**(*s*) ne sera exécutée que lorsqu'aucunes opérations **P**(*s*) et / ou **V**(*s*) ne sont en cours et la valeur de $s > 0$.

2.a : une opération **V**(*s*) n'est suspendue que lorsqu'une opération **P**(*s*) ou **V**(*s*) est en cours.
b. une opération **P**(*s*) n'est suspendue que lorsqu'une opération **P**(*s*) ou **V**(*s*) est en cours ou la valeur de $s = 0$.

3.a : une opération **V**(*s*) ne sera jamais suspendue indéfiniment.
b : une opération **P**(*s*) sera suspendue indéfiniment uniquement si il existe un moment où la valeur de *s* restera $=0$ indéfiniment (c-à-d que plus aucune opération **V**(*s*) ne sera exécutée).

## 2.2. RESOLUTION DE PROBLEMES DE SYNCHRONISATION AU MOYEN DE SEMAPHORES.

Nous allons d'abord montrer comment résoudre quelques problèmes généraux de synchronisation. Ensuite, nous présenterons la résolution des trois versions du "problème des représentants".

N.B. : rappelons que la première version consiste à faire communiquer des processus *rep(i)* et *recense(j)* au moyen d'un buffer de taille 1. La deuxième version est identique à la première, excepté que le buffer est de taille N et est implémenté en "anneau". Enfin, la dernière version présente une implémentation du buffer au moyen de listes. Ces trois versions ont donc des spécifications identiques : seul l'implémentation du buffer diffère !

## 2.2.1. RESOLUTION DE PROBLEMES GENERAUX.

a) *Problème 1* : ce problème fait intervenir deux opérations *op1* et *op2*. Les contraintes de synchronisation dans ce problème se résument à imposer un ordre d'exécution entre *op1* et *op2* : l'opération *op2* devra être exécutée APRES l'opération *op1*. L'opération *op2* est donc soumise à la condition de synchronisation suivante : "*op1* a déja été exécutée".

*Comment implémenter cette condition de synchronisation avec des sémaphores ?*

Rappelons que - *P(s)* implémente l'opération suivante : $\{s > 0\}$ $s := s-1$, c-à-d qu'elle provoque une suspension du processus tant que $s = 0$, puis décrémente $s$ de 1 (en exclusion mutuelle avec des opérations P(*s*) ou V(*s*)).
- V(*s*) implémente l'opération $s := s+1$, ce qui peut entrainer le réveil de processus suspendu sur un P(*s*).

La condition de synchronisation de *op2* peut être implémentée au moyen d'une variable entière $s$, ayant la signification suivante : $s = 0 \Longleftrightarrow$ *op1* n'a pas encore été exécutée ($s \in \{0,1\}$). Il suffit alors que l'opération *op2* soit précédée d'une opération qui suspend le processus tant que $s = 0$. De même, l'opération *op1* sera suivie d'une opération qui signale que *op1* a été exécutée, c-à-d qui affecte à $s$ la valeur 1.

| | |
|---|---|
| | $\{s > 0\} \Rightarrow$ **P(s)** |
| *op1* ; | *op2* ; |
| $s := 1 \Rightarrow$ **V(s)** | |

Initialement, la valeur de $s$ est égale à 0. En voyant comment fonctionne les opérations P(*s*) et V(*s*), il est clair que l'opération qui précède *op2* est un P(*s*) et celle qui suit *op1* est un V(*s*).

*b) Problème 2* : ce problème fait intervenir un ensemble *OP* d'opérations. Les contraintes de synchronisation dans ce problème consistent à imposer une exclusion mutuelle sur *OP*, c-à-d que les opérations de *OP* soient exécutées strictement les unes après les autres. Une opération de *OP* est donc soumise à la condition d'exclusion mutuelle suivante : "aucune opération de *OP* n'est en cours". (*)

*Comment implémenter cette condition d'exclusion mutuelle ?*

Il suffit de déclarer une variable entière *s*, avec la signification suivante : $s = o \Longleftrightarrow$une opération de *OP* est en cours ($s \in \{0,1\}$). Une opération de *OP* sera alors précédée d'une opération qui suspend le processus tant que $s = 0$, puis lorsque $s \neq 0$ (c-à-d $s = 1$),signale qu'une opération de *OP* est en cours, en mettant 0 dans *s* (c-à-d $s := s-1$).

De même, une opération de *OP* sera suivie d'une opération signalant que plus aucune opération de *OP* n'est en cours (c-à-d $s := s+1$)

$$\left.\begin{array}{l}\{s > 0\}\\ s := s-1\end{array}\right\} \Longrightarrow P(s)$$

*op* ;

$$\left.\begin{array}{l}s := s+1\end{array}\right\} \Longrightarrow V(s)$$

Initialement, la valeur de *s* est égale à 1.

*c) Problème 3* : ce problème fait intervenir deux types d'opérations que l'on désignera par *TOP1* et *TOP2*. Les contraintes de synchronisation consiste à imposer une exécution strictement ordonnée de l'ensemble des opérations de ces types, en alternant successivement les opérations de type *TOP1* et *TOP2*. De plus, on impose que la première opération qui sera exécutée est une opération de type *TOP1*.

(*) Ce problème est bien connu sous le nom de "l'exclusion mutuelle de sections critiques" [2]

Une opération de type *TOP1* est donc soumise aux contraintes de synchronisation suivantes :

1. pas d'opérations de type *TOP1* ou *TOP2* en cours.
2. aucune opération n'a encore été exécutée ou la dernière opération exécutée est une opération de type *TOP2*.

Une opération de type *TOP2* est soumise aux contraintes de synchronisation suivantes :

1. idem que pour *TOP1*.
2. des opérations ont déja été exécutées et la dernière est de type *TOP1*.

*Comment implémenter ces conditions (d'exclusion mutuelle / synchronisation) ?*

On va déclarer 2 variables entières *s1* et *s2*, avec les significations suivantes : (*)

*s1* = *0* $\Longleftrightarrow$ les conditions 1. et / ou 2. de *TOP1* sont fausses.

*s2* = *0* $\Longleftrightarrow$ les conditions 1. et / ou 2. de *TOP2* sont fausses.

($s1, s2 \in \{0,1\}$).

N.B. : $s1 = 1 \Rightarrow s2 = 0$ (mais $s2 = 0 \not\Rightarrow s1 = 1$)

Une opération de type *TOP1* sera alors précédée d'une opération qui suspend le processus tant que $s1 = 0$, puis lorsque $s1 \neq 0$ (c-à-d $s1 = 1$, et $s2 = 0$) signale qu'une opération de type *TOP1* est en cours, en mettant 0 dans *s1* ($s1 := s1-1$)

Une opération de type *TOP1* sera alors suivie d'une opération qui signale qu'il n'y a plus d'opérations de type *TOP1* et *TOP2* en cours, et que la dernière opération qui a été exécutée est de type *TOP1* ($\Rightarrow s1 = 0$ et $s2 = 1$ ; $\Rightarrow s2 := s2+1$).

Le raisonnement est similaire pour une opération de type *TOP2*.

(*) Ici, 2 variables sont nécessaires car les deux types d'opérations ont chacun leurs contraintes de synchronisation.

```
{ s1 > 0 }                          { s2 > 0 }
s1 := s1-1   ⟩⇒ P(s1)               s2 := s2-1   ⟩⇒ P(s2)
TOP1 ;                              TOP2 ;
s2 := s2+1   ⟩⇒ V(s2)               s1 := s1+1   ⟩⇒ V(s1)
```

Initialement : *s1 = 1* et *s2 = 0*.

*2.2.2. RESOLUTION D'EXEMPLES CONCRETS.*

Nous allons résoudre au moyen de sémaphores, les problèmes de synchronisation rencontrés dans les trois versions du "problème des représentants".

*a) Première version (cfr I.13-18)* : Rappelons que nous avons trois types d'opérations sur les variables communes : les types "ecrire", "deposer" et "prendre". Les contraintes de synchronisation sur ces types d'opérations peuvent être décomposés en deux groupes :

1. Les opérations *ecrire* doivent être mutuellement exclusives.
2. L'ensemble des opérations *deposer* et *prendre* doivent être exécutées strictement les unes après les autres, en alternant successivement les opérations de type *deposer* et de type *prendre*. De plus, la première opération qui sera exécutée doit être une opération *deposer*.

Les contraintes du premier groupe sont équivalentes à celles définies pour le problème 2 (cfr II.4). Il suffit donc de déclarer un sémaphore, par exemple *ecrir* et de l'initialiser à 1. Chaque opération *ecrire* sera alors précédée d'un *P(ecrir)* et suivie d'un *V(ecrir)*.

```
P(ecrir) ;
write(commande-2000,y) ;  ⟩⇒ ecrire
V(ecrir) ;
```

Rappelons que les opérations *ecrire* sont exécutées par les processus *recense(j)* $(1 \leq j \leq c)$.

Les contraintes du second groupe sont équivalentes à celles définies pour le problème 3 (cfr II.4-5-6), où *TOP1* est équivalent au type d'opération *deposer* et *TOP2* au type *prendre*. Il suffit donc de déclarer deux sémaphores, par exemple *dep* et *pre* initialisés respectivement à 1 et 0.

| | |
|---|---|
| **P**(*dep*) ; | **P**(*pre*) ; |
| *buffer* := *x* ; | *x* := *buffer* ; |
| **V**(*pre*) ; | **V**(*dep*) ; |

*Remarque* : la variable booléenne *s* qui permettait de synchroniser les opérations *deposer* et *prendre* (cfr I.15) n'a pas d'utilité ici car elle est remplacée par les deux sémaphores *pre* et *dep*.

*b) Deuxième version (cfr I.18-22)* : Nous avons de nouveau les types d'opérations *ecrire*, *deposer* et *prendre*. Les contraintes de synchronisation sont de nouveau décomposées en deux groupes :

1. idem que pour la première version et la résolution est la même.
2. nous avons des contraintes du type "exclusion mutuelle" :
   - exclusion mutuelle des opérations *prendre*.
   - exclusion mutuelle des opérations *deposer*.

   Nous avons aussi des contraintes du type "condition de synchronisation" :
   - opération *prendre* : $j-i > 0$ (*i* étant le nombre d'opérations *prendre* qui ont été exécutées et *j* le nombre d'opérations *deposer* qui ont été exécutées). $(j-i)$ signifie en fait le nombre de cellules *pleines* actuellement dans le buffer.
   - opération *deposer* : $N-(j-i) > 0$ ; $N-(j-i)$ est le nombre de cellules *vides*.

Il suffit donc de déclarer 2 sémaphores *pre* et *dep* avec les significations suivantes :

*pre* : la valeur de ce sémaphore représentera le nombre de cellules pleines dans le buffer (initialement : $pre=0$).

*dep* : la valeur de ce sémaphore représentera le nombre de cellules vides dans le buffer (initialement : $dep=N$).

Une opération *prendre* devra alors être précédée d'une opération qui suspend le processus tant que *pre = 0*. Dès que *pre > 0*, il faut signaler qu'on réserve une cellule pleine, c-à-d en décrémentant *pre* de 1.

Une opération *prendre* devra alors être suivie d'une opération qui signale qu'il y a une cellule vide en plus, c-à-d en incrémentant *dep* de 1.

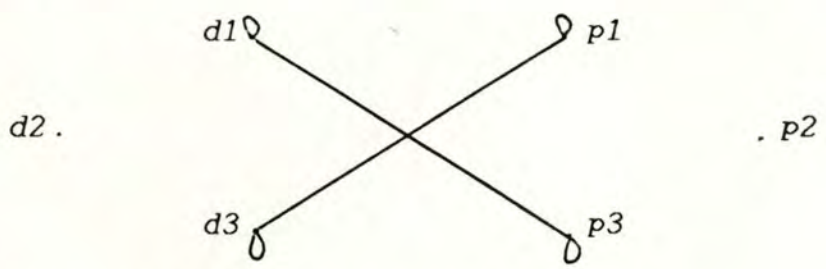
Le raisonnement est similaire pour une opération *deposer*, mais en remplaçant *dep* par *pre* et vice-versa.

Les sémaphores *pre* et *dep* permettent de respecter les conditions de synchronisation. Il reste cependant à respecter les contraintes d'exclusion mutuelle : on utilise pour cela les sémaphores *pmutex* et *dmutex* qui assureront respectivement l'exclusion mutuelle des opérations *prendre* et *deposer* (valeurs initiales : *pmutex, dmutex = 1*).

```
        P(dep) ;                          P(pre) ;
        P(dmutex) ;                       P(pmutex) ;
        buffer [ j mod N ] := x ;         x := buffer [ i mod N ] ;
deposer     j := j+1 ;                        i := i+1 ;          prendre
        V(dmutex) ;                       V(pmutex) ;
        V(pre) ;                          V(dep) ;
```

*c) Troisième version (cfr I.22-28)* : dans cette version, les opérations *deposer* et *prendre* ont été chacunes décomposées en 3 sous-opérations : *d1, d2* et *d3* pour *deposer* et *p1, p2* et *p3* pour *prendre*.

Les contraintes d'exclusion mutuelle sont imposées d'une part, sur l'ensemble des opérations *d1* et *p3*, et d'autre part sur les opérations *p1* et *d3*.




(graphe d'exclusion mutuelle).

Les contraintes de condition de synchronisation sont de deux types :
- une opération *d1* ne pourra être exécutée que lorsque la liste *VD* est non vide, c-à-d que le nombre de cellules vides > 0.
- une opération *p1* ne pourra être exécutée que lorsque la liste *PL* est non vide, c-à-d que le nombre de cellules pleines > 0.

*Nous allons implémenter ces contraintes au moyen de quatre sémaphores :*

- *VDMUTEX* : pour l'exclusion mutuelle des opérations *d1* et *p3*.
- *PLMUTEX* : pour l'exclusion mutuelle des opérations *p1* et *d3*.
- *VID* : ce sémaphore implémentera la condition de synchronisation des opérations *d1*. Sa valeur indiquera le nombre de cellules vides (valeur initiale : N). Une opération *d1* sera précédée d'un **P**(*VID*) et une opération *p3* sera suivie d'un **V**(*VID*), car *p3* ajoute une cellule vide.
- *PLE* : ce sémaphore implémentera la condition de synchronisation des opérations *p1*. Sa valeur indiquera le nombre de cellules pleines (valeur initiale : 0). Une opération *p1* sera précédée d'un **P**(*PLE*) et une opération *d3* sera suivie d'un **V**(*PLE*).

**deposer**

```
    P(VID) ;
    P(VDMUTEX) ;
d1  ( v := VD ; VD := T[VD] ;
    V(VDMUTEX) ;
d2  ( buffer[v] := x ;
    P(PLMUTEX) ;
    if   PL = N
d3  ( then PL := v else T[fPL] := v ;
    T[v] := N ; fPL := v ;
    V(PLMUTEX) ;
    V(PLE) ;
```

**prendre**

```
    P(PLE) ;
    P(PLMUTEX) ;
    p := PL ; PL := T[PL] ;   ) p1
    V(PLMUTEX) ;
    x := buffer[p] ;          ) p2
    P(VDMUTEX) ;
    if   VD = N
    then VD := p else T[fVD] := p ;   ) p3
    T[p] := N ; fVD := p ;
    V(VDMUTEX) ;
    V(VID) ;
```

*Remarques* : - rappelons que les deux codes ci-dessus se trouveront respectivement dans des processus *rep(i)* et *recense(j)*.
- dans cette implémentation, les opérations *d2* et *p2* peuvent être concurrentes : on pourra avoir jusqu'à N opérations *d2* et / ou *p2* en cours à un instant donné.
- l'ordre des opérations **P***(VID)* et **P***(VDMUTEX)* est capital : il est indispensable de n'obtenir l'exclusion mutuelle sur la liste *VD* que lorsqu'on est assuré que *VD* est non vide, car sinon on risque d'être bloqué sur un **P***(VID)*, tout en maintenant l'exclusion mutuelle sur *VD* (⇒bloquage indéfini !).

## 2.3. INCONVENIENTS DU SEMAPHORE.

Nous avons vu que le principe de cette primitive était fort simple : un sémaphore est en fait une sorte de "*distributeur de jetons*", l'opération **P** consistant à demander un jeton au distributeur (il faut évidemment attendre que le distributeur ne soit pas vide) et l'opération **V** consistant à mettre un jeton dans le distributeur.

Cette simplicité donne au sémaphore deux grandes qualités :

- tout d'abord, un caractère de "non-spécificité" : la simplicité du sémaphore permet d'utiliser cette primitive dans la plupart des problèmes de synchronisation, et pas seulement dans une classe particulière de problèmes. On trouvera d'ailleurs dans [2], plusieurs problèmes, très différents les uns des autres, incluant des notions de priorités, de droit d'accès,... qui ont été résolus au moyen de sémaphores.
- ensuite, une souplesse dans l'utilisation du sémaphore. La simplicité de cette primitive laisse en effet au programmeur une grande liberté dans la façon dont il va résoudre les problèmes de synchronisation, permettant ainsi d'obtenir des solutions très performantes (cfr la résolution de la "troisième version", p II.9, où on peut avoir jusqu'à N opérations *d2* et / ou *p2* qui manipulent le buffer en même temps).

Cette simplicité est malheureusement aussi la cause de plusieurs inconvénients, qui font du sémaphore une primitive peu adéquate dans un langage de programmation de "haut-niveau".

On constate tout d'abord des inconvénients dus au fait qu'avec des sémaphores, les processus peuvent accéder directement aux ressources communes, sans passer par un "contrôleur" quelconque.

*) Un des premiers inconvénients de cet "accès direct" est la non-transparence des ressources : en effet, les processus doivent connaître exactement la façon dont sont implémentées les ressources, car ils les manipulent directement. Le moindre changement dans leur implémentation entraîne alors des changements dans tous les processus qui les manipulent.

Exemple : si on reprend les trois versions du "problème des représentants", on constate que les spécifications de chacunes d'elles sont les mêmes, excepté la façon dont est implémenté le buffer. Etant donné que les processus *rep(i)* et *recense(j)* ont accès directement au buffer, il est clair que passer d'une version à une autre entraînera "inutilement" un changement dans tous les processus *rep(i)* et *recense(j)*. "Inutilement" car le fait de passer d'une version à une autre ne devrait pas entraîner de changement au niveau des processus, puisque les spécifications du problème n'ont pas changées.

*) Un second inconvénient est l'absence de contrôle sur qui accède aux ressources communes et ce que l'on fait sur ces ressources. En effet, l'accès direct permet une politique du type "n'importe qui fait n'importe quoi".

*) Enfin, un dernier inconvénient du à l'accès direct est la décentralisation de la synchronisation : on constate en effet que la synchronisation des opérations est réalisée séparément dans chaque processus. Cela signifie que l'ensemble des opérations seront correctement synchronisées uniquement si chaque processus a correctement synchronisé ses propres opérations avec le reste des opérations. Une synchronisation décentralisée est très difficile à réaliser, surtout si les processus sont "codés" par différentes personnes.

Il y a bien sûr encore d'autres inconvénients :

*) Par exemple, la simplicité des opérations **P** et **V** rend difficile la compréhension d'un programme concurrent synchronisé au moyen de sémaphore : "à quoi sert tel sémaphore", "quelle est sa signification

exacte",... Tous ces renseignements doivent alors être enoncés explicitement par le ou les programmeurs.

*) De même, cette simplicité ne permet pas de réaliser, avant exécution, des vérifications automatiques sur la validité des accès : par exemple, vérifier que tout accès à des ressources communes se réalise bien en exclusion mutuelle avec toute opération pouvant y interférer.

Le lecteur interressé trouvera dans [2], [8], [10], [15] des critiques interessantes sur les caractéristiques du sémaphore par rapport à d'autres primitives.

Tous les inconvénients cités précédemment ont conduit à la proposition de primitives beaucoup plus "structurées" et plus adéquates aux contraintes de la programmation de "haut-niveau".

Le chapitre suivant est consacré à la présentation d'une de ces primitives, la plus connue qui est le **MONITOR**.

# CHAPITRE TROIS

# Le monitor

## 3.1. Présentation du monitor.

### 3.1.1. Principe.
### 3.1.2. Description du monitor.
#### 3.1.2.1. Syntaxe.
#### 3.1.2.2. Sémantique.
### 3.1.3. Quelques précisions...
#### 3.1.3.1. Appel d'une procédure d'un monitor.
#### 3.1.3.2. Appels de procédure imbriqués.
#### 3.1.3.3. Comportement d'un monitor.

## 3.2. Problème du monitor : la sécurité aux dépens des performances.

### 3.2.1. Résolution des versions du "problème des représentants"
### 3.2.2. Conclusion : le monitor "concurrent".

Le chapitre précédent nous a montré les inconvénients du sémaphore, inconvénients dus surtout au fait qu'avec cette primitive, les processus pouvaient accéder directement aux ressources communes.

Nous nous proposons dans ce chapitre de présenter le *monitor*, primitive de synchronisation de "haut-niveau" proposée par HOARE [9] et HANSEN [16], dont l'objectif premier était de supprimer, ou en tout cas de limiter, les inconvénients du sémaphore.

Nous donnerons d'abord une présentation générale du monitor (principe, syntaxe, sémantique). Ensuite, nous verrons au moyen de quelques exemples (les 3 versions du "problème des représentants") le principal problème du monitor : c'est que si, en effet, il supprime les gros inconvénients du sémaphore, il supprime malheureusement aussi le principal avantage de cette primitive qui était son efficacité.

Nous terminerons enfin par une conclusion sur le monitor, qui nous amènera à proposer une nouvelle version de cette primitive, le monitor concurrent, dont l'objectif est d'allier à la fois les avantages du monitor (sécurité, abstraction des données,...) et les avantages du sémaphore (performance, efficacité,...).

## 3.1. PRESENTATION DU MONITOR.

### *3.1.1. PRINCIPE.*

L'objectif du monitor est d'éviter que des processus aient accès directement aux ressources communes. Pour cela, on divise l'ensemble des ressources communes en *classes*, c-à-d en sous - ensembles de ressources communes disjoints deux à deux, et sur chaque classe est définit un ensemble de procédures, n'ayant accès qu'aux ressources de la classe.

Un processus pourra alors accéder aux ressources d'une classe uniquement par l'intermédiaire des procédures définies sur celle - ci, c-à-d par l'appel d'une de ces procédures. Un processus ne peut donc plus accéder directement aux ressources communes : il ne peut les manipuler qu'au travers des procédures définies sur la classe contenant les ressources. Une classe, et les procédures qui lui sont associées est appelée un MONITOR.

La synchronisation des procédures dans un monitor est réalisée de la manière suivante :

1. L'exclusion mutuelle des procédures d'une même classe est *garantie*, c-à-d que les procédures d'une même classe seront exécutées stictement les unes après les autres.
2. Des instructions du type *"wait(B)"* où *B* est une expression booléenne quelconque, permettent d'implémenter les conditions de synchronisation des procédures.

Nous allons maintenant décrire de façon précise la notation utilisée pour décrire un monitor, et la signification de cette notation.

### *3.1.2. DESCRIPTION DU MONITOR.*

#### 3.1.2.1. Syntaxe.

Nous allons montrer comment décrire un monitor (c-à-d un ensemble de ressources + un ensemble de procédures sur ces ressources) au moyen de la notation proposée par HOARE [9].

Cette notation fut en fait empruntée à celle de SIMULA67 [17] pour décrire le concept de "classe".

```
monitorname : monitor
   begin < déclaration des ressources du monitor > ;

            procedure procname
            (in liste des paramètres "entrée" out..."sortie") ;
               begin . . . < procedure body > . . . end ;
            .
            .
            .
            procedure procname
            (in liste des paramètres "entrée" out..."sortie") ;
               begin . . . < procedure body > . . . end ;

         < initialisation des ressources du monitor >
   end ;
```

3.1.2.2. Sémantique.

Les différents composants pour décrire un monitor sont les suivants :

1. l'identificateur du monitor : *monitorname*.

2. un mot réservé spécifiant que l'on a affaire à une description de monitor : ***monitor***.

3. la description des ressources du monitor. Celle-ci se fait en utilisant une notation de type PASCAL.

4. la description des procédures du monitor : pour chaque procédure, on décrit les paramètres formels (paramètres "entrée" ou par valeur, et paramètres "sortie" ou par adresse) et le corps de la procédure (variables locales à la procédure + liste d'opérations de la procédure). Ces procédures sont accessibles aux processus, au moyen de l'instruction suivante :

*monitorname.procname (liste des paramètres effectifs)*

qui est un simple appel de procédure, spécifiant le nom de la procédure (*monitorname.procname*) et les paramètres effectifs d'entrée / sortie.

*Remarques* : - rappelons que les ressources d'un monitor ne sont accessibles que par les procédures de celui-ci, et que les procédures d'un monitor n'ont accès qu'aux ressources de ce monitor.

- l'exclusion mutuelle des procédures d'un monitor est garantie (c-à-d qu'elle n'est pas à charge du programmeur) : une procédure d'un monitor ne sera donc exécutée que lorsque aucune autre procédure de ce monitor n'est en cours !

5. un code d'initialisation des ressources du monitor, qui sera exécuté avant tout appel de procédures.

6. Les conditions de synchronisation des procédures d'un monitor pourront être implémentées grâce à deux notations spéciales :

- tout d'abord, on pourra déclarer dans un monitor des variables de type CONDITION, de la manière suivante :

  *variablename* : **COND** [ < *expression booléenne* > ]

  N.B. : l'expression booléenne ne peut référencer que les variables du monitor.
- Ensuite, seule l'opération suivante sera possible sur des variables CONDITION : **wait**(*a*). Si *a* est une variable de type **COND**, alors l'exécution de cette opération par une procédure a pour effet de suspendre la procédure aussi longtemps que *(a:B)* est faux. *(a:B (*)* étant une notation pour désigner l'expression booléenne associée à la variable *a*).

*Remarques* : - une procédure suspendue n'est plus en cours, et donc l'exclusion mutuelle sur le monitor est relâchée, ce qui permet l'entrée en cours d'une autre procédure.
- les variables de type COND ne peuvent être déclarées qu'à l'intérieur d'un monitor.

### *3.1.3. QUELQUES PRECISIONS...*

Avant de passer aux exemples utilisant des monitors, il nous a parut utile de préciser certaines notions.

#### 3.1.3.1. Appel d'une procédure de monitor.

Nous avons vu que l'appel d'une procédure d'un monitor se réalisait au moyen de l'opération *"monitorname.procname (liste param. effectifs)"*.

Cette opération a exactement le même effet qu'un simple appel de procédure, c-à-d :

a) création d'une copie des données locales de la procédure *procname*, et association de cette copie au contexte du processus exécutant l'appel.

(*) Exemple : si l'on a déclaré : *VIDE* : **COND** [ *N=0* ] , alors la notation *(VIDE : B)* désigne l'expression *N=0*.

b) exécution de la procédure *procname*, dans le contexte du processus exécutant l'appel.

c) suppression de la copie des données locales de *procname* et retour à l'opération suivant l'appel.

Les points a) et c) n'ont rien de particulier et sont commmuns à tous les appels de procédures. Par contre, la réalisation (ou l'implémentation) du point b) devra offrir au programmeur les garanties suivantes :

1.a.: l'exécution d'une procédure ne pourra être réalisée que lorsque le monitor est "libre" (c-à-d qu'aucune autre procédure du même monitor n'est en cours d'exécution).

1.b. : l'exécution d'une opération **wait**(*a*) devra suspendre l'exécution de la procédure aussi longtemps que *(a:B)* est faux.

2. : l'exécution d'une procédure ne sera suspendue que si le monitor n'"est pas libre" ou l'exécution est sur un **wait**(*a*) avec *(a:B)* qui est faux.

3. l'exécution d'une procédure ne sera suspendue indéfiniment que si le monitor n'est plus jamais "libre" (c-à-d qu'une autre procédure est en cours indéfiniment dans le monitor) ou bien l'exécution est sur un **wait**(*a*), avec *(a:B)* qui restera faux pour toujours.

On remarquera que ces 3 garanties assurent la synchronisation correcte (cfr I.28-29) des procédures d'un monitor. Nous verrons dans le chapitre 5 comment implémenter au moyen de sémaphores, les procédures d'un monitor afin que ces 3 garanties soient respectées.

### 3.1.3.2. Appels de procédures imbriqués.

Jusqu'à présent, nous avons vu que, seul des processus pouvaient appeler les procédures d'un monitor. Il est cependant aussi interessant de permettre des appels de procédures imbriqués, c-à-d permettre que les procédures d'un monitor puissent appeler les procédures d'autres monitors.

Lorsque la procédure *P* d'un monitor *M* appelle la procédure *Q* d'un monitor *N*, nous considérons que la procédure appelante (c-à d *P*) n'est plus en cours dans *M* pendant le temps que dure l'exécution

de la procédure appelée (c-à-d *Q*) : un appel de procédure sera donc précédé d'une opération qui signale que la procédure sort du monitor (et donc relâche l'exclusion mutuelle) et sera suivi d'une opération qui signale que la procédure veut à nouveau rentrer dans le monitor (demande d'exclusion mutuelle).

N.B. : . les opérations qui précèdent et suivent un appel de procédure sont invisibles du programmeur.
. les paramètres effectifs d'un appel de procédure réalisé dans une procédure *P* ne feront référence qu'à des variables locales de *P*.
. le lecteur interessé par les appels imbriqués (notamment par la possibilité d'options alternatives sur le comportement de la procédure appelante) se réfèrera aux articles parus dans [18], [19], [20], [21].

### 3.1.3.3. Comportement d'un monitor.

On peut résumer le comportement d'un monitor de la façon suivante : lorsque la procédure d'un monitor est activée (c-à-d est appelée par un processus ou par une procédure de monitor), l'exécution de la procédure est suspendue tant que le monitor est "occupé" (c-à-d qu'une procédure est en cours dans le monitor). Lorsque le monitor est "libre" (aucune procédure n'y est en cours), la procédure est exécutée (et le monitor devient "occupé"). Au cours de son exécution, un des cas suivants se présentera (on suppose que le temps d'exécution des procédures est fini !) :

a) L'exécution se termine : dans ce cas, le monitor redevient "libre" ($\Longrightarrow$une autre procédure peut entrer dans le monitor).
b) L'exécution arrive sur un appel de procédure : dans ce cas, le monitor redevient "libre" et l'appel de procédure est exécuté. A la fin de cet appel, la procédure sera suspendue tant que le monitor est "occupé". Lorsque le monitor est "libre", l'exécution de la procédure est reprise et le monitor devient "occupé".
c) L'exécution arrive sur un *wait(a)* :
si *(a:B)* est vrai : aucun effet et le monitor reste "occupé".

si *(a:B)* est faux : l'excution de la procédure est suspendue (et le monitor redevient "libre"). L'exécution sera reprise dès que *(a:B)* est vrai et le monitor est "libre". Lors de la reprise, le monitor devient "occupé".

## 3.2. PROBLEME DU MONITOR : LA SECURITE AU DEPENS DES PERFORMANCES.

Nous allons montrer comment résoudre les trois versions du "problème des représentants" et comparer ces résolutions avec celles utilisant le sémaphore : nous terminerons cette section par une conclusion sur le monitor qui nous amènera à proposer une nouvelle version du monitor, le *monitor concurrent*.

### *3.2.1. RESOLUTION DES VERSIONS DU "PROBLEME DES REPRESENTANTS".*

#### a) Première version (cfr I.13-18)

Nous allons diviser l'ensemble des ressources communes en deux classes disjointes :

- une classe contenant le *buffer* et la variable booléenne *s*. Sur cette classe seront définies les procédures *deposer* et *prendre*.
- une classe contenant le fichier *commande-2000*, sur laquelle est définie la procédure *ecrire*.

La première classe sera décrite par le monitor *"buffer"*; la seconde par le monitor *"com2000"*.

```
buffer : monitor
   begin buffer : com ; s : boolean ;
         OK deposer : COND [ s=false ] ;        { variable de type
         OK prendre : COND [ s=true ] ;          CONDITION }

      procedure deposer (in x : com) ;
        begin wait (OK deposer) ;          { suspend la procédure
              buffer := x ; s := true        aussi longtemps que
        end ;                               (OK deposer : B) est faux }
      procedure prendre (out x : com) ;
        begin wait (OK prendre) ;
              x := buffer ; s := false
        end ;
       s := false                  { code d'initialisation }
   end ;

com2000 : monitor
   begin commande-2000 : file of com ;

      procedure ecrire (in y : com) ;
        begin write (commande-2000, y) end ;
          { initialisation de commande-2000 }

   end ;
```

Le code d'un processus *recense(j)* sera alors le suivant :

```
process recense ;
   var   y : com ;
   begin while true do begin buffer.prendre(y) ;
                             com2000.ecrire(y)
                       end
   end ;
```

Code d'un processus *rep(i)* :

```
process rep ;
   var   commande : file of com ; x : com ;
   begin while not (eof (commande))
         do begin read (commande,x) ;
                  if   x.montant ⩾ 2000
                  then buffer.deposer(x)
            end
   end ;
```

N.B. : Nous avons laissé tomber les opérations d'initialisation et de cloture du fichier *"commande"*.

Comparaison avec le sémaphore :

*) Une des premières différences que l'on remarque avec le sémaphore est l'absence, dans le code des processus, de références directes aux variables communes. En effet, les processus *rep(i)* et *recense(j)* ne référencent plus directement les variables *buffer*, *s* et *commande-2000* mais plutôt y accèdent au moyen de procédures prédéfinies. La façon dont sont représentées concretement les ressources communes est donc CACHEE aux processus. Les processus ne connaissent en fait d'une ressource que son *interface*, c-à-d les procédures définies dessus et les spécifications de ces procédures, spécifications décrites sans faire référence aux variables représentant la ressource mais plutôt en utilisant une représentation abstraite de la ressource. Une telle approche est appelée *ABSTRACT DATA TYPES*, c-à-d "types de données abstraits". Le lecteur intéressé par les types de données abstraits et leur spécification se réfèrera à [22], [23], [24].

L'avantage de cette abstraction des données est évident : étant donné que les processus ne savent rien de l'implémentation d'une ressource, un changement dans celle-ci n'entrainera AUCUNE modification du code des processus, car l'interface n'a pas changé.

Nous verrons concrètement cela lorsque l'on passera à la deuxième version du "problème des représentants", où seule l'implémentation du buffer a changé : les spécifications abstraites des procédures *deposer* et *prendre* n'ont, elles, pas changé et donc le codes des processus *rep(i)* et *recense(j)* ne doit subir aucune modification.

*) Une seconde différence est la centralisation de la synchronisation : en effet, la synchronisation correcte des opérations *prendre* et *deposer* est réalisée à un seul endroit, qui est le monitor *buffer*. Avec le sémaphore, chaque processus était responsable de la synchronisation de ses opérations avec les opérations des autres processus. Outre la difficulté de réaliser une telle synchronisation, le risque de synchronisation incorrect est très grand. Par exemple, un processus "mal-intentionné" peut facilement bloquer indéfiniment tout un système en oubliant de réaliser une opération **V**(*s*).

*) Une troisième différence avec le sémaphore est que la politique du "n'importe qui fait n'importe quoi" n'a plus cours. En effet, les seules opérations permises sur une ressource commune seront les procédures prédéfinies sur le monitor contenant la ressource. Par exemple, seules les opérations *prendre* et *deposer* pourront être exécutées sur le buffer, ce qui n'était pas le cas avec le sémaphore.

De plus, un contrôle sur qui appelle les procédures d'un monitor est possible : il suffit de stocker dans le monitor un ensemble d'informations spécifiant "qui peut faire quoi". Ainsi, lors d'un appel de procédure, il suffit de vérifier si le processus exécutant l'appel a bien le droit d'exécuter cette procédure.

*) Le monitor a encore d'autres avantages. Par exemple, une synchronisation réalisée au moyen de monitor est plus facile à comprendre que si elle était réalisée avec des sémaphores : synchronisation centralisée, opérations plus "évoluées" (ex : **wait**(*a*)),...

Un autre avantage est la possibilité de vérifier certaines contraintes sur la validité des accès, lors de la compilation.

Par exemple, on peut vérifier que :

- un processus n'accède qu'à ses données locales, et aux procédures des monitors.
- les procédures d'un monitor n'accèdent qu'aux variables du monitor, et bien sûr à leurs variables locales.
- les paramètres effectifs d'un appel de procédure réalisé par une procédure *P* ne font référence qu'à des variables locales de *P*.

*) En ce qui concerne les *performances*, il n'y a pas de différences : qu'il s'agisse de la résolution avec le sémaphore ou celle avec le monitor, le degré de concurrence (*) est le même, pour cette première version : au maximum, on aura deux opérations en cours en même temps : une opération *ecrire* et une opération *prendre / déposer*.

N.B. : si on avait déclaré un seul monitor, contenant le buffer et le fichier commande-2000, le degré de concurrence aurait été plus faible (une seule opération en cours au maximum), car les opérations *ecrire*, *deposer* et *prendre* auraient été mutuellement exclusives (cfr sémantique d'un monitor, III.3).

b) Deuxième version (cfr I.18-22)

La seule chose qui change par rapport à la première version est l'implémentation du buffer. Nous allons donc aller modifier le monitor *buffer* de la façon suivante :

```
buffer : monitor
   begin buffer : array [0..N-1] of com ;
         I,J : integer ;
         OK deposer : COND [J-I < N] ;  { nbr. cellules vides > 0 }
         OK prendre : COND [J-I > 0] ;  { nbr. cellules pleines > 0 }

      procedure deposer (in x : com) ;
         begin wait (OK deposer) ;
               buffer [J mod N] := x ; J := J+1
         end ;
      procedure prendre (out x : com) ;
         begin wait (OK prendre) ;
               x := buffer [I mod N] ; I := I+1
         end ;
      I := 0 ; J := 0
   end ;
```

(*) c-à-d le nombre d'opérations en cours sur les ressources communes.

Comparaison avec le sémaphore :

*) On remarque tout d'abord concrètement l'avantage de l'abstraction des données offert par le monitor. En effet, les modifications dans le buffer n'entraineront AUCUNES modifications dans le code des processus *rep(i)* et *recense(j)* car l'interface du buffer est restée la même (mêmes procédures, mêmes paramètres et mêmes spécifications).

Si on compare avec la résolution de la version 2 utilisant le sémaphore, on observe les mêmes avantages, que ceux cités en III.9-10 : centralisation de la synchronisation, contrôle de l'accès aux données ("qui peut faire quoi"), compréhension plus facile d'un programme, vérification possible de certaines contraintes sur la validité des accès à la compilation...

*) Par contre, on observe dans cette résolution une baisse de performance : en effet, dans la deuxième version résolue avec des sémaphores, le degré de concurrence était de **3** : une opération *deposer* pouvait s'exécuter en concurrence avec une opération *prendre*, et ces deux opérations peuvent s'exécuter en même temps qu'une opération *ecrire*.

Avec la version résolue au moyen de monitor, le degré de concurrence n'est plus que de **2** : en effet, une opération *deposer* ne peut plus être exécutée en concurrence avec une opération *prendre*, car les procédures d'un même monitor sont mutuellement exclusives.

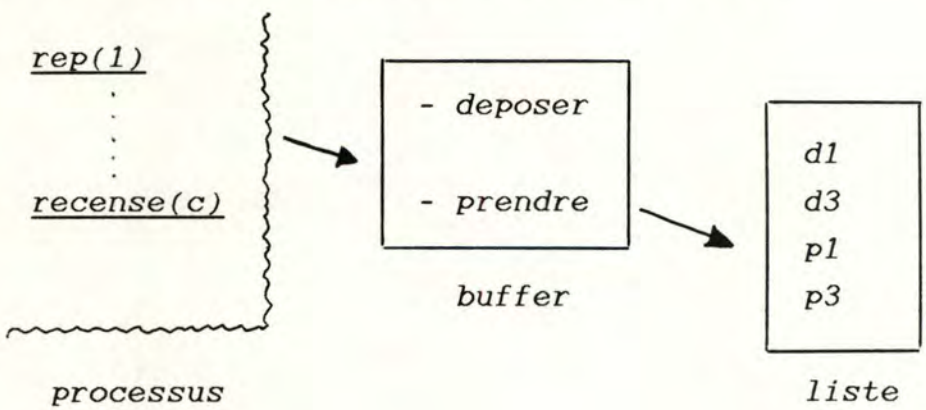
N.B. : on constatera que cette baisse de performance est inévitable. En effet, pour obtenir un degré de concurrence de **3**, on peut
- soit déclarer les procédures *prendre* et *deposer* dans des monitors distincts, afin qu'elles puissent être concurrentes. Cette solution est impossible car ces deux procédures accèdent à la même ressource.
- soit ne plus déclarer le buffer dans un monitor et y accéder directement. Cette solution est également rejetée car elle violerait les contraintes sur la validité des accès aux ressources communes.

c) Troisième version (cfr I.22-28)

Dans cette version le buffer sera implémenté au moyen de listes (*PL* et *VD*); les opérations *deposer* seront décomposées en trois sous-opérations *d1*, *d2*, *d3* et les opérations *prendre* seront elles aussi décomposées en trois sous-opérations *p1*, *p2*, *p3*.

Afin que ces changements soient cachés aux processus *rep(i)* et *recense(j)*, nous allons conserver le monitor *buffer* et ses procédures *deposer* et *prendre*. De plus, nous définirons le monitor *liste* représentant les listes *PL* et *VD*, sur lequel seront définies les procédures manipulant ces listes (c-à-d *d1*, *d3*, *p1*, *p3*).

N.B. : le monitor *buffer* sera accessible aux processus *rep(i)* et *recense(j)*; par contre, le monitor *liste* ne sera accessible qu'aux procédures du monitor *buffer*, ceci afin de cacher les détails sur l'implémentation du buffer.




```
buffer : monitor
   begin buffer : array [ 0..N-1 ] of com ;

      procedure deposer (in x : com) ;
                v : integer ;    { variable locale de deposer }
         begin liste.d1(v) ;
               buffer [ v ] := x ;          { d2 }
               liste.d3(v)    { appel de procédure }
         end ;
      procedure prendre (out x : com) ;
                p : integer ;
         begin liste.p1(p) ;
               x := buffer [ p ] ;          { p2 }
               liste.p3(p)
         end
   end ;
```

```
liste : monitor
   begin  T : array [ 0..N-1 ] of integer ;
          PL, fPL, VD, fVD : integer ;
          OK.d1 : COND [ VD≠N ] ;
          OK.p1 : COND [ PL≠N ] ;

      procedure d1 (out v : integer) ;
          begin wait (OK.d1) ; v := VD ; VD := T[VD]  end ;

      procedure p1 (out p : integer) ;
          begin wait (OK.p1) ; p := PL ; PL := T[PL]  end;

      procedure d3 (in v : integer) ;
          begin if   PL ≠ N
                then T[fPL]  := v   else PL := v ;
                T[v]  := N ; fPL := v
          end ;

      procedure p3 (in p : integer) ;
          begin if   VD ≠ N
                then T[fVD]  := p   else VD := p ;
                T[p]  := N ; fVD := p
          end ;

      { initialiser les variables VD, PL, fVD et T }

   end ;
```

Comparaison avec le sémaphore :

De nouveau, on remarque tous les avantages observés dans les deux premières versions, c-à-d abstraction des données, centralisation de la synchronisation, contrôle de "qui fait quoi", compréhension plus facile,...

On remarquera aussi dans cette version que l'on a deux niveaux d'abstraction :l'un au niveau des processus (la représentation concrète du buffer et du fichier commande-2000 leur est cachée). L'autre au niveau du monitor *buffer* : la représentation concrète des deux listes lui est cachée. Le fait d'implémenter les deux

listes *PL* et *VD* au moyen de pointeurs (comme en PASCAL) par exemple n'entrainerait AUCUNE modification dans le monitor *buffer*.

*Que se passe-t'il au niveau des performances ?*

Dans la première version, la résolution au moyen du monitor n'entrainait aucune baisse de performances. Dans la deuxième version, on observe déja une légère baisse du degré de concurrence. Avec la résolution de la troisième version, on peut dire que cela devient catastrophique !

En effet, dans la résolution avec des sémaphore, le degré de concurrence était de **N+1** : on pouvait avoir jusqu'à **N** opérations appartenant à $\{d1, d2, d3, p1, p2, p3\}$ en cours simultanément avec une opération *ecrire*.

soit
- *N* opérations $\in \{d2, p2\}$
- *N-1* opérations $\in \{d2, p2\}$ + 1 opération $\in \{d1, d3, p1, p3\}$
- *N-2* opérations $\in \{d2, p2\}$ + 1 opération $\in \{d1, p3\}$ + 1 opération $\in \{p1, d3\}$

Avec le monitor, le degré de concurrence n'est plus que de **3** ! En effet, parmi les opérations de $\{d1, d2, d3, p1, p2, p3\}$, deux au plus peuvent être en cours à un instant donné : une opération $\in \{d2, p2\}$ et une opération $\in \{d1, p3, d3, p1\}$.

On constatera que dans n'importe quelles solutions,

- les opérations *d2* et *p2* devront être mutuellement exclusives car elles doivent appartenir au même monitor.
- les opérations *d1*, *d3*, *p1*, *p3* devront être mutuellement exclusives car elles manipulent une variable commune (le tableau *T*) et donc seront déclarées dans le même monitor.

Dans n'importe quelle solution, on n'arrivera donc jamais à un degré de concurrence plus grand que **3** : une opération $\in \{d2, p2\}$ + une opération $\in \{d1, d3, p1, p3\}$ + une opération *ecrire*.

Cette situation sera d'autant plus ennuyeuse au niveau des performances que la valeur de *N* est grande et / ou le temps d'exécution des opérations $\{d2, p2\}$ est important.

### *3.2.2. CONCLUSION : LE MONITOR "CONCURRENT".*

D'après les exemples vus, il est clair que le monitor est un outil beaucoup plus agréable à utiliser que le sémaphore : il permet entre autres d'obtenir :

*) une connaissance abstraite des ressources (on sait ce que l'on peut faire sur les ressources mais on ne sait pas comment on le fait). De plus, les appels imbriqués entre monitors permettent d'obtenir une hiérarchie de niveaux d'abstraction (cfr III.14), appelée aussi "couches d'abstraction" (*), la couche la plus haute se situant au niveau des processus.

*) une synchronisation centralisée.

*) un contrôle sur { ce que l'on fait sur les ressources.
                    { "qui" accède aux ressources.

.
.
.

Par contre, on a vu que l'utilisation du monitor entrainait des baisses de performances par rapport au sémaphore.

Avec le sémaphore, le programmeur réalisait la synchronisation comme il l'entendait et donc pouvait parvenir à un degré de concurrence optimal. Ce n'est plus le cas avec le monitor, car toutes les procédures d'un monitor sont automatiquement mutuellement exclusives. Seules des procédures appartenant à différents monitors peuvent être exécutées en concurrence. Cette exclusion mutuelle automatique nous parait beaucoup trop restrictive !

En effet, il arrive souvent que l'on soit obligé de déclarer dans le même monitor (n'oublions pas que les procédures d'un monitor n'accèdent qu'aux variables de leur monitor, et que les monitors sont disjoints 2 à 2) des procédures qui pourraient être exécutées en concurrence, et donc ces procédures seront inutilement mutuellement exclusives !

(*) Des systèmes célèbres configurés en "couches d'abstractions" sont par exemple le T.H.E [13] et le système O.S.I [25].

*Exemples* : . on peut avoir d'abord le cas où plusieurs exemplaires de la même procédure peuvent être exécutés en concurrence. Par exemple, une procédure *Read* définie dans le monitor *data base*, qui consiste à lire la transaction courrante de la base de donnée. Plusieurs *read* peuvent être en cours simultanément, mais la loi d'exclusion mutuelle automatique impose inutilement une exclusion mutuelle sur ces *read*.

. on peut avoir aussi deux procédures *P1* et *P2* qui n'ont aucunes variables en commun (et donc peuvent être concurrentes). Mais supposons que *P1* et *P2* aient chacunes des variables en commun avec une procédure *G*.
Dans ce cas : *P1* et *G* doivent être dans le même monitor
*P2* et *G* doivent être dans le même monitor
$\Longrightarrow$ *P1* et *P2* doivent être dans le même monitor et seront donc inutilement mutuellement exclusives.

. on peut avoir deux procédures *d2* et *p2* qui accèdent au même vecteur, mais chacunes à une cellule distincte : *d2* et *p2* peuvent donc être concurrentes mais étant donné qu'elles ont une variable en commun, elles doivent être déclarées dans le même monitor ($\Longrightarrow$ exclusion mutuelle inutile).
N.B. : ce cas est présent dans la 3° version (cfr III.15)

Toutes ces observations nous ont amené à proposer une nouvelle version du monitor permettant de conserver à la fois les avantages du monitor, et le principal avantage du sémaphore, qui est la liberté laissée au programmeur pour résoudre les problèmes de synchronisation, ce qui permet d'obtenir de hauts degrés de concurrence.

Cette nouvelle version, appelée MONITOR CONCURRENT, sera décrite dans le chapitre suivant. Le dernier chapitre sera consacré à l'implémentation, au moyen de sémaphores, du *monitor* et du *monitor concurrent*.

A titre d'information, voici quelques langages de programmation concurrente dont les primitives de synchronisation sont basées sur le principe du monitor : *concurrent PASCAL* [1], *MESA* [26], *MODULA* [27], *MODULA-2* [28], *ADA* [29], *DP* [3] et *SR* [5].

# CHAPITRE QUATRE

# Le monitor concurrent

## 4.1. Principe du monitor concurrent.

## 4.2. Présentation du monitor concurrent.

### 4.2.1. Syntaxe / Sémantique.

#### 4.2.1.1. Nouvelles notations.

#### 4.2.1.2. Comportement du monitor concurrent.

### 4.2.2. Critères d'exclusion mutuelle.

## 4.3. Exemples.

### 4.3.1. Problèmes des représentants.

#### 4.3.1.1. Première version.

#### 4.3.1.2. Deuxième version.

#### 4.3.1.3. Troisième version.

Ce chapitre est essentiellement consacré à la présentation du *monitor concurrent*, qui permet à la fois d'obtenir les qualités du monitor (abstraction, sécurité,...) et celles du sémaphore (performance, efficacité,...).

Après avoir expliqué le principe de cette nouvelle primitive, nous en donnerons une présentation complète, c-à-d la syntaxe et la sémantique des notations qui sont introduites par rapport au monitor.

Nous cloturerons ce chapitre par la résolution des trois versions du "problème des représentants" au moyen du monitor concurrent.

## 4.1. PRINCIPE DU MONITOR CONCURRENT.

Comme nous l'avons déja signalé plusieurs fois, le monitor concurrent doit allier à la fois les avantages du monitor et du sémaphore. Pour ce faire, nous allons d'abord conserver dans le monitor concurrent les caractéristiques du monitor, qui en faisaient ses principales qualités. Ces caractéristiques sont :

*) division de l'ensemble des ressources communes en classes disjointes 2 à 2, et définition sur chaque classe d'un ensemble de procédures.

*) accès aux ressources d'une classe uniquement par les procédures définies sur cette classe.

*) les procédures d'une classe n'ont accès qu'aux ressources de leur classe.

*) utilisation possible d'opérations de type *wait*, permettant l'ordonnancement de procédures d'un monitor. Ces opérations ne peuvent être utilisées que dans les monitors.

*) possibilités d'appels imbriqués de procédures.

Toutes ces caractéristiques nous permettent de conserver les avantages du monitor :

- abstration des ressources (+ couches d'abstraction).
- centralisation de la synchronisation.
- contrôle sur "qui fait quoi".
- compréhension plus facile d'un programme concurrent.
- possibilité de vérifier automatiquement, lors de la compilation, le respect de certaines contraintes sur la validité des accès.

Le principal inconvénient du monitor est la règle d'exclusion mutuelle automatique des procédures d'un même monitor : cela signifie que seules des procédures appartenant à des monitors différents peuvent être concurrentes. Nous avons vu en III.17 que cette règle est beaucoup trop restrictive, et avait comme conséquence d'entrainer une baisse de performance par rapport au sémaphore. Nous n'allons donc plus reprendre cette règle dans le monitor concurrent, et permettre ainsi que des procédures d'un même monitor puissent être concurrentes. Ce sera donc au programmeur à spécifier dans un monitor, les contraintes d'exclusion mutuelle sur les procédures de ce monitor.

Un monitor concurrent a donc les mêmes caractéristiques qu'un monitor "conventionnel", excepté la règle d'exclusion mutuelle, qui est supprimée : il allie donc à la fois les avantages du monitor, et la souplesse du sémaphore.

Le seul inconvénient est le risque d'erreur : étant donné que c'est au programmeur à spécifier dans un monitor si oui ou non deux procédures seront concurrentes, celui-ci peut se tromper : permettre que deux procédures soient concurrentes alors qu'en fait elles auraient dû être mutuellement exclusives.

Ce risque d'erreur peut être diminué en réduisant fortement le rôle du programmeur, c-à-d en automatisant la tâche de calculer si oui ou non 2 procédures s'interferent. Cette tâche ne peut cependant être entièrement automatisée, car pour certains types d'interférences (sur des variables indicées par exemple), le rôle du programmeur est nécessaire.

*Exemple* : soit les procédures *P1* et *P2*.

*P1 : a[i] := m ;* *P2 : a[j] := n*

(*a* est une variable commune !)

Afin de déterminer si oui ou non il y a interférences entre *P1* et *P2*, il faut voir si au moment où sont exécutées ces deux procédures, *i* et *j* peuvent avoir une valeur identique (si oui, alors il y a interférence).

Ce problème, d'une façon générale, ne peut être résolu par un programme, et donc l'intervention du programmeur est nécessaire.

## 4.2. PRESENTATION DU MONITOR CONCURRENT.

Cette section présente la syntaxe et la sémantique du monitor concurrent, ou plus exactement des nouvelles notations introduites par rapport au monitor "conventionnel". Ensuite, nous examinerons les critères permettant de décider si deux procédures d'un même monitor doivent être mutuellement exclusives ou peuvent être concurrentes.

### *4.2.1. SYNTAXE / SEMANTIQUE.*

#### 4.2.1.1. Nouvelles notations.

Rappelons que le monitor concurrent diffère seulement du monitor "conventionnel" par le fait que le programmeur spécifie à l'intérieur du monitor les contraintes d'exclusion mutuelle sur les procédures du monitor. Dans un monitor conventionnel, chaque procédure est automatiquement mutuellement exclusive avec elle-même et avec toutes les autres procédures du monitor.

La syntaxe d'un monitor concurrent sera donc identique à celle d'un monitor conventionnel (cfr III.2), à l'exception de 2 points :

a) le mot réservé **monitor** est remplacé par **monitor concurrent**.

b) une clause spéciale peut être associée à chaque procédure du monitor. Cette clause a le format suivant :

**excludes** [ *p1, p2, ... pn* ]

*et spécifie que la procédure à laquelle est associée la* clause doit être mutuellement exclusive avec les procédures *p1, ...pn.*

```
monitorname : monitor concurrent
   begin < déclaration des ressources du monitor > ;

      procedure procname (in liste.paramètres "entrée" out..
                                .."sortie")
                  { excludes [p1, ... pn] } ;
      .  begin   . . .   end ;
      :
      procedure procname (in ..............  out ..............)
                  { excludes [g1, ... gn] } ;
         begin   . . .   end ;

         < initialisation des ressources du monitor >
   end ;
```

*Remarques* : a) la clause **excludes** est optionelle : son absence devant une procédure signifie que la procédure ne doit être mutuellement exclusive ni avec elle-même, ni avec les autres procédures du monitor.

b) les spécifications des contraintes d'exclusion mutuelle doivent être cohérentes, c-à-d que si la clause **excludes** [ *p1...pn* ] est associée à la procédure *procname*, alors il faut que :

- les *p1...pn* soient des noms de procédures distincts, appartenant au même monitor que *procname*.
- une clause **excludes** [ ...*procname*... ] soit associée à chaque procédure *pi*, ( $\forall$ $1 \leq i \leq n$). Cela est du au caractère symétrique de la relation d'exclusion mutuelle.

c) on peut remarquer que le monitor "conventionnel" est en fait un cas particulier de monitor concurrent, où à chaque procédure serait associée une clause **excludes** [ *p1...pn* ] , les *p1...pn* étant chaque fois les noms de toutes les procédures du monitor.

4.2.1.2. Comportement du monitor concurrent.

Comme nous avons décrit en p. III.6-7 le comportement d'un monitor "conventionnel", nous allons décrire celui du monitor concurrent.

Supposons qu'une procédure de nom *procname*, associée à une clause **excludes** [ *p1...pn* ] soit activée (c-à-d soit appelée par un processus ou par une procédure de monitor) : l'exécution de *procname* sera suspendue tant que des procédures de noms *p1...pn* sont en cours.

Lorsqu'il n'y a plus aucune procédure de ces noms en cours, la procédure *procname* est exécutée (il y a donc une procédure *procname* de plus en cours).

Au cours de son exécution, un des cas suivants se produira : (on suppose que le temps d'exécution de *procname* est fini !).

a) L'exécution se termine : dans ce cas, il y a une procédure *procname* de moins en cours.

b) L'exécution arrive sur un appel de procédure : dans ce cas, il y a une procédure *procname* de moins en cours, et l'appel de procédure est exécuté. A la fin de cet appel, la procédure *procname* sera suspendue tant qu'il y aura des procédures de noms *p1...pn* en cours. Lorsqu'il n'y a plus aucune procédure de ces noms en cours, l'exécution de la procédure est reprise ( $\Longrightarrow$ il y a une procédure *procname* de plus en cours).

c) L'exécution arrive sur un *wait(a)* :

si *(a:B)* est vrai : aucun effet et *procname* reste en cours

si *(a:B)* est faux : l'exécution de la procédure est suspendue ( $\Longrightarrow$ une procédure *procname* de moins en cours). L'exécution sera reprise dès que *(a:B)* est vrai et il n'y a plus aucune procédure *p1...pn* en cours. Lors de la reprise, il y aura une procédure *procname* de plus en cours.

*4.2.2. CRITERES D'EXCLUSION MUTUELLE.*

Sur quels critères le programmeur va-t-il se baser pour déterminer si deux procédures doivent être mutuellement exclusives ?

Nous avons vu dans le chapitre 1 que deux procédures (ou opérations) devaient être mutuellement exclusives si elles pouvaient s'interférer, c-à-d que l'une pouvait modifier des variables référencées par l'autre procédure.

N.B. : le fait qu'une procédure puisse modifier les variables référencées par un *wait(a)* (c-à-d les variables référencées par *(a:B)*) est aussi une interférence.

*Quelques exemples d'interférences entre deux procédures P1 et P2*

1.

| P1 | P2 |
|---|---|
| . | . |
| . | . |
| **op1** ( *i := i+1 ;* | **op2** ( *a := i ;* |
| . | . |
| . | . |

L'exécution concurrente de **op1** et **op2** peut conduire à des résultats incohérents. En effet, supposons que la référence à la variable *i* dans **op2** soit implémentée de la façon suivante :

*LOADH1 i ;* { la 1° moitié de *i* est placée dans la 1° moitié de l'accumulateur } .

*LOADH2 i ;* { idem mais avec la 2° moitié } .

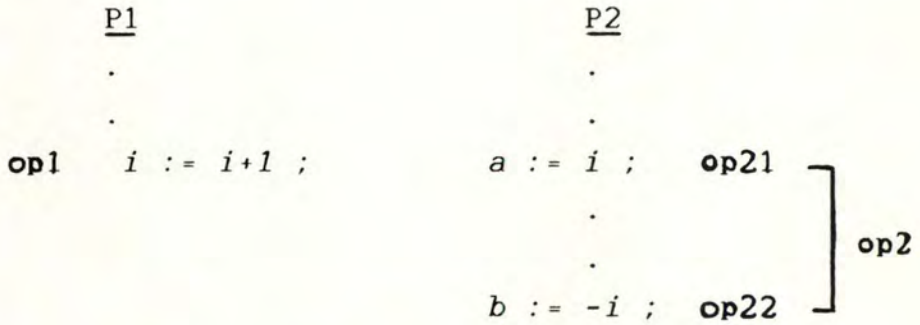
Afin de garantir la cohérence de cette double instruction, il faut que la valeur de *i* ne change pas entre l'exécution de *LOADH1 i ;* et *LOADH2 i ;* ($\Longrightarrow$ **op1** ne peut être exécuté en même temps que **op2**).

Cette exécution concurrente sera évitée en imposant une contrainte d'exclusion mutuelle entre P1 et P2.

N.B. : on pourrait imaginer un autre système de monitor concurrent où les contraintes d'exclusion mutuelle ne seraient plus spécifiées au niveau des procédures mais au niveau des opérations dans une procédure. Ce choix n'a pourtant pas été

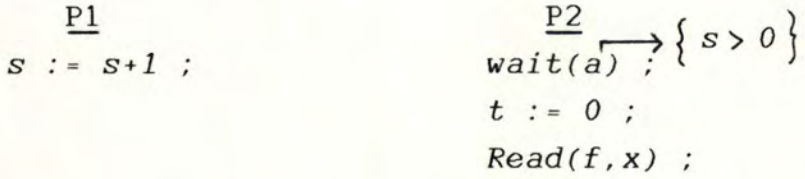
retenu car il nécessiterait une notation et une implémentation plus complexe. De plus, la participation du programmeur serait accrue, ce qui augmenterait le risque d'erreur (cfr IV.2)

2.



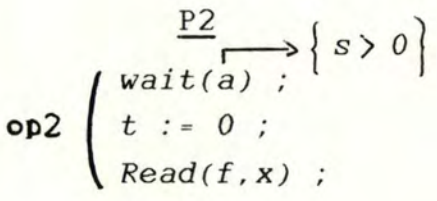

De nouveau, P1 et P2 devront être mutuellemnt exclusives afin d'éviter une exécution concurrente de op1 et op2.

3.




L'exécution de $s := s+1$ doit être réalisée en exclusion mutuelle avec l'évaluation de $\{ s > 0 \} \Longrightarrow$ P1 et P2 devront être mutuellement exclusives.

4.




On a la même interférence qu'en 3. mais en plus P1 peut rendre faux la condition $\{ s > 0 \}$ : l'exécution de $s := s-1$ devra donc être réalisée en exclusion mutuelle avec **op2** $\Longrightarrow$ P1 et P2 devront être mutuellement exclusives.

## 4.3. EXEMPLES.

Nous allons résoudre avec le monitor concurrent les trois versions du "problème des représentants" : nous verrons ainsi concrètement que le monitor concurrent permet de conserver les avantages

du monitor conventionnel, et en même temps d'obtenir les mêmes performances que celles observées avec le sémaphore.

Nous nous limiterons à ces trois exemples, bien qu'il en existe encore d'autres (par exemple, le problème du "Readers-Writers", décrit en [ 30 ], et résolu avec un monitor conventionnel, par HOARE [ 9 ]) qui montrent l'avantage du monitor concurrent sur le monitor conventionnel.

### *4.3.1. PROBLEME DES REPRESENTANTS.*

#### 4.3.1.1. Première version (cfr I.13-18)

Pour cet exemple, il n'y a aucun avantage pour le programmeur à utiliser le monitor concurrent. En effet, son utilisation n'est avantageuse que si dans un monitor conventionnel, il y a des procédures qui pourraient être concurrentes.

Or, on constate (cfr III.8) que :

- dans le monitor *com2000* (contient une procédure:*ecrire*), la procédure *ecrire* doit être mutuellement exclusive avec elle-même (car interférences sur *commande-2000*).
- dans le monitor *buffer* (contient 2 procédures : *deposer* et *prendre*),
  - . la procédure *deposer* doit être mutuellement exclusive avec elle-même, car interférences sur *buffer* et *s*.
  - . la procédure *prendre* doit être mutuellement exclusive avec elle-même, car interférences sur *s*.
  - . les procédures *deposer* et *prendre* doivent être mutuellement exclusives (car interférences sur *buffer* et *s*) et de plus, l'ordonnancement de ces deux procédures élimine toutes possibilités d'exécution concurrente.

Le programmeur peut cependant sans problèmes utiliser un monitor concurrent. Pour cela, on garde les monitors décrits en III.8 avec les modifications suivantes :

1) dans le monitor *buffer*,
   - on remplace ***monitor*** par ***monitor concurrent***.
   - on associe à la procédure *deposer* la clause

     **excludes** [ *deposer, prendre* ]

- on associe à la procédure *prendre* la clause
  **excludes** [ *deposer, prendre* ]

2) dans le monitor *com2000,*
   - on remplace **monitor** par **monitor concurrent**.
   - on associe à la procédure *ecrire* la clause
     **excludes** [ *ecrire* ]

*Remarques* : *) que la première version soit résolue avec le sémaphore, le monitor conventionnel ou le monitor concurrent, le degré de concurrence est de **2**.

*) on notera que le passage d'un monitor conventionnel à un monitor concurrent, si il n'entraîne pas toujours d'avantages, n'entraîne cependant aucun inconvénient (si ce n'est la spécification des contraintes d'exclusion mutuelle). En effet, toutes les qualités du monitor conventionnel, décrites dans le chapitre précédent, se retrouvent dans le monitor concurrent. Cela est du au fait que le monitor conventionnel n'est en fait qu'un cas particulier de monitor concurrent (cfr IV.4). L'inverse, (concurrent $\longrightarrow$ conventionnel) peut par contre entraîner de graves baisses de performances.

*) le fait de déclarer les procédures *deposer*, *prendre* et *ecrire* dans un même monitor concurrent n'entrainerait aucune baisse de performances : en effet, la procédure *ecrire* pourrait toujours être exécutée en concurrence avec une procédure *deposer* ou *prendre*.
Ce n'est pas le cas avec le monitor conventionnel, où les 3 procédures seraient automatiquement mutuellement exclusives.
Nous préferons cependant définir 2 monitors concurrents, afin de préserver une transparence entre le *buffer* et le fichier *commande-2000* : le concepteur du *buffer* n'a pas besoin de connaître les détails d'implémentation du fichier *commande-2000*, et vice-versa.

4.3.1.2. Deuxième version (cfr I.18-22)

Examinons la résolution de cette version avec le monitor conventionnel (cfr III.11), et regardons, pour chaque monitor défini, si il y a des procédures qui pourraient être concurrentes. Si c'est le cas, alors l'utilisation d'un monitor concurrent s'averera plus efficace.

.) *monitor com2000* (une procédure *ecrire*) :

Les procédures *ecrire* doivent être mutuellement exclusives, car elles s'interfèrent l'une l'autre (sur le fichier *commande-2000*).
L'utilisation d'un monitor concurrent pour le fichier *commande-2000* ne présente donc aucun avantage (et aucun inconvénient non plus !).

.) *monitor buffer* (deux procédures : *deposer* et *prendre*).

Entre 2 procédures *deposer* : ces procédures s'interfèrent sur le *buffer* et sur $J \Longrightarrow$une procédure *deposer* doit être mutuellement exclusive avec elle-même.

Entre 2 procédures *prendre* : ces procédures s'interfèrent sur $I \Longrightarrow$ une procédure *prendre* doit être mutuellement exclusive avec elle-même.

Entre 1 procédure *deposer* et 1 procédure *prendre* :

```
deposer ; ↱{ J-I< N }          prendre ; ↱{ J-I > 0 }
wait (OK deposer) ;            wait (OK prendre) ;
buffer [ J mod N ] := x ;      x := buffer [ I mod N ] ;
J := J+1 ;                     I := I+1 ;
```

Les seules interférences entre ces deux procédures sont au niveau des opérations *wait* :

- une procédure *prendre* peut modifier (avec $I := I+1$) les variables référencées par *wait (OK deposer)* (c-à-d $J-I < N$). Ces modifications ne rendront cependant jamais fausse l'expression $J-I < N$.

- une procédure *deposer* peut modifier (avec *J := J+1*) les variables référencées par *wait (OK prendre)* (c-à-d *J-I > 0*). Ces modifications ne rendront cependant jamais fausse l'expression *J-I > 0*.

Cela signifie par exemple que pendant que la procédure *deposer* est occupée à accéder à la variable *I* lors de l'évaluation de l'expression *(J-I < N)*, la procédure *prendre* peut être en train d'exécuter *I := I+1*, pouvant ainsi entraîner des résultats imprévisibles si l'accès à la variable *I* n'est pas "atomique" (EX : LOADH1 I; LOADH2 I).

Pour éviter ce type d'interférences, il est donc nécessaire que les procédures *deposer* et *prendre* soient aussi mutuellement exclusives. Dans cette version, le monitor concurrent ne présente donc à première vue aucun avantage par rapport au monitor conventionnel, car dans les 2 cas, le degré de concurrence est de 2, c-à-d un de moins qu'avec le sémaphore.

En effet, avec le sémaphore, une opération *deposer* pouvait être concurrente avec une opération *prendre* (cfr II.8) :

| deposer | prendre |
|---|---|
| *P(dep) ;* | *P(pre) ;* |
| *P(dmutex) ;* | *P(pmutex) ;* |
| *buffer[ j mod N ] := x ;* | *x := buffer[ i mod N ] ;* |
| *j := j+1 ;* | *i := i+1 ;* |
| *V(dmutex) ;* | *V(pmutex) ;* |
| *V(pre) ;* | *V(dep) ;* |

On constate ici qu'il n'y a plus aucunes interférence entre une opération *deposer* et une opération *prendre*, car l'évaluation des conditions de synchronisation des deux opérations (c-à-d *J-I < N* et *J-I > 0*) n'est plus réalisée explicitement mais est remplacée par les opérations *P(dep)* et *P(pre)*.

En utilisant ce principe, nous pouvons trouver une implémentation de la 2° version avec le monitor concurrent, où les procédures *deposer* et *prendre* pourraient être concurrentes. Pour cela, il suffit de déclarer un monitor qui implémentera les sémaphores *dep* et *pre*, accompagnés de leurs opérations *P* et *V*.

```
semaph : monitor concurrent ;
   begin dep, pre : integer ;
         OK dep : COND [ dep > 0 ] ;
         OK pre : COND [ pre > 0 ] ;

      procedure Pdep excludes [ Pdep, Vdep ] ;
         begin wait (OK dep) ; dep := dep-1 end ;        { P(dep) }

      procedure Vdep excludes [ Pdep, Vdep ] ;
         begin dep := dep+1 end ;                        { V(dep) }

      procedure Ppre excludes [ Ppre, Vpre ] ;           { P(pre) }
         begin wait (OK pre) ; pre := pre-1 end ;

      procedure Vpre excludes [ Ppre, Vpre ] ;           { V(pre) }
         begin pre := pre := pre+1 end ;

     dep := N ; pre := 0                  { initialisation }

   end ;
```

Le monitor *buffer* a alors la forme suivante:

```
buffer : monitor concurrent ;
   begin buffer : array [ 0..N-1 ] of com ;     I, J : integer ;

      procedure deposer (in x : com) excludes [ deposer ] ;
         begin semaph.Pdep ;              { P(dep) }
               buffer [ J mod N ] := x ;          J := J+1
               semaph.Vpre              { V(pre) }
         end ;

      procedure prendre (out x : com ) excludes [ prendre ] ;
         begin semaph.Ppre ;
               x := buffer [ I mod N ] ;        I := I+1
               semaph.Vdep
         end ;

     I := 0 ; J := 0
   end ;
```

*Remarques* :

*) Rappelons que lorsqu'un appel de procédure est réalisé dans le monitor *buffer* (par exemple *Semaph.Pdep*), la procédure appelante (ici, *deposer*) quitte le monitor juste avant l'appel, permettant à une autre procédure *deposer* d'entrer dans le monitor. Lorsque l'appel est terminé, la procédure demandera à réentrer dans le monitor : elle sera donc suspendue tant que des procédures avec lesquelles elle est mutuellement exclusive (ici, *deposer*) sont en cours.

*) L'exclusion mutuelle des opérations *deposer* (ou plutôt des procédures *deposer*) est garantie par la clause **excludes** [ *deposer* ]

Idem pour les procédures *prendre*.

*) Avec cette implémentation, les procédures *deposer* et *prendre* peuvent être concurrentes sans risque d'interférences. On a donc atteint les mêmes performances qu'avec le sémaphore, tout en préservant les avantages du monitor. Par exemple, il suffit de regarder le code d'un processus *rep(i)*, lorsqu'il dépose une commande *x* dans le buffer.

avec : le monitor concurrent

```
buffer.deposer(x) ;
```

sémaphore

```
P(dep) ;
P(dmutex) ;
buffer [ j mod N ] := x ;
j := j+1 ;
V(dmutex) ;
V(pre) ;
```

*) La réalisation de l'implémentation que l'on vient de voir (c-à-d un monitor *buffer* et un monitor *semaph*) avec des monitors conventionnels au lieu de monitors concurrents aurait donné un degré de concurrence moindre : les procédures *prendre* et *deposer* n'auraient pas pu être concurrentes car exclusion mutuelle automatique !

4.3.1.3. Troisième version (cfr I.22-28)

C'est bien sûr dans cette version que le monitor concurrent est vraiment avantageux. Rappelons qu'avec la résolution utilisant le monitor conventionnel, le degré de concurrence était de 3 (cfr III.15) : . une opération *ecrire* dans le monitor *com2000*.

. une opération $\in$ { deposer (d2), prendre (p2) } dans le monitor *buffer*.

. une opération $\in$ { d1, d3, p1, p3 } dans le monitor *liste*.

*Or, on constate que :*

1. dans le monitor *buffer* (deux procédures *deposer* et *prendre*), plusieurs procédures *deposer* (ou *d2*) peuvent être exécutées en concurrence avec plusieurs procédures *prendre* (p2). En effet, il n'y a aucune interférence entre ces procédures.

2. dans le monitor *liste* (4 procédures : *d1, d3, p1, p3*), une procédure $\in$ { d1, p3 } peut être exécutée en concurrence avec une procédure $\in$ { p1, d3 }. (cfr I.25)

Nous allons donc :

- utiliser pour le *buffer* un monitor concurrent dans lequel aucune clause **excludes** ne sera utilisée (car il n'y a aucune contrainte d'exclusion mutuelle !). On gardera donc le monitor *buffer* décrit en III.13 avec la seule modification suivante : on remplaçe **monitor** par **monitor concurrent**.
- utiliser pour *liste* un monitor concurrent dans lequel on spécifiera les contraintes d'exclusion mutuelle nécessaire. On gardera donc le monitor *liste* décrit en III.14 avec les modifications suivantes :
  . remplacer **monitor** par **monitor concurrent**.
  . associer à la procédure *d1* la clause **excludes** [ *d1, p3* ]
  *d3* la clause **excludes** [ *p1, d3* ]
  *p1* la clause **excludes** [ *p1, d3* ]
  *p3* la clause **excludes** [ *d1, p3* ]

Cet exemple montre bien que le monitor concurrent allie à la fois les avantages du monitor conventionnel et l'efficacité du sémaphore. On constate en effet que toutes les qualités du monitor

conventionnel se retrouvent dans cet exemple (couches d'abstraction, synchronisation centralisée,...), mais en plus le degré de concurrence de cette résolution est le même que celui observé avec le sémaphore, c-à-d **N+1** (cfr III.15). Avec le monitor conventionnel, ce degré ne dépassait pas 3.

Jusqu'à présent, nous avons présenté le monitor (conventionnel / concurrent) sous l'angle "utilisation", c-à-d toutes les informations nécessaires au(x) programmeur(s) pour utiliser un monitor : son principe, sa syntaxe, sa sémantique et son comportement.

Nous avons vu aussi comment résoudre des exemples concrets avec un monitor.

Nous allons maintenant nous interesser à l'implémentation des procédures d'un monitor, afin qu'elles soient synchronisées correctement. Cette implémentation sera réalisée au moyen de sémaphores, primitives de synchronisation de bas-niveau décrites au chapitre II (cette implémentation sera évidemment par définition cachée au programmeur !).

Nous allons décrire dans le dernier chapitre, l'implémentation des 2 modèles de monitor :

*1) Le monitor conventionnel* (décrit au chapitre III) : dans ce modèle, toutes les procédures sont mutuellement exclusives. Ce modèle est identifié par le mot-clè **monitor**.

*2) Le monitor concurrent* (décrit dans ce chapitre) : dans ce modèle, c'est le programmeur qui spécifie les contraintes d'exclusion mutuelle, au moyen des clauses **excludes**. Afin d'éviter des résultats incohérents, le programmeur "doit" imposer une exclusion mutuelle sur tout couple de procédures pour lesquelles il existe des interférences. Ce modèle est identifié par le mot-clé **monitor concurrent**.

*N.B.* : nous essaierons de proposer une implémentation "intelligente" c-à-d qui soit capable de déceler des cas où l'implémentation peut être très simplifiée. Par exemple, un monitor concurrent où toutes les procédures doivent être mutuellement exclusives est un de ces cas : ce monitor peut en effet être implémenté comme un monitor conventionnel.

# CHAPITRE CINQ

# Implémentation

## 5.1. Implémentation d'un monitor conventionnel.

5.1.1. Modifications à introduire dans les procédures.
5.1.2. Annexes.
    5.1.2.1. Justification des modifications.
    5.1.2.2. Codage PASCAL des modifications.
    5.1.2.3. Exemple.
5.1.3. Remarques.
    5.1.3.1. Simplification de l'implémentation.
    5.1.3.2. Opérations WAIT avec variables locales.
    5.1.3.3. Opérations WAIT avec "priorité".

## 5.2. Implémentation d'un monitor concurrent.

5.2.1. Modification à introduire dans les procédures.
5.2.2. Annexes.
    5.2.2.1. Justification de ces modifications.
    5.2.2.2. Codage PASCAL des modifications.
    5.2.2.3. Exemple.
5.2.3. Remarques.
    5.2.3.1. Simplification de l'implémentation.
    5.2.3.2. Problèmes posés par le monitor concurrent.

Nous allons présenter dans ce chapitre l'implémentation des deux modèles de monitor : monitor *conventionnel* et monitor *concurrent*.

L'implémentation d'un monitor signifiera ici l'ensemble des modifications à réaliser sur les procédures du monitor afin que celles-ci soient correctement synchronisées. Le sémaphore sera notre primitive de synchronisation utilisée dans les implémentations.

Nous avons choisi cette primitive en raison des avantages qu'elle présente (cfr II.10) : simplicité syntaxique et sémantique ($\Rightarrow$ facile à implémenter) ; souplesse d'utilisation ($\Rightarrow$ permet des performances interessantes), usage à caractère général,...

## 5.1. IMPLEMENTATION D'UN MONITOR CONVENTIONNEL.

Nous présenterons d'abord les modifications à réaliser sur les procédures d'un monitor conventionnel pour qu'elles soient synchronisées correctement. La justification et le codage PASCAL de ces modifications se trouvera en annexe.

Nous terminerons cette section par quelques remarques sur l'implémentation : nous montrerons par exemple que l'implémentation peut être très simplifiéé (et donc plus efficace) dans certains cas particuliers.

N.B. : Pour l'implémentation d'un monitor conventionnel, nous nous sommes directement inspirés de [ 31 ] (*). Nous avons cependant choisi une manière différente pour expliquer et justifier cette implémentation.

### *5.1.1. MODIFICATIONS A INTRODUIRE DANS LES PROCEDURES.*

Afin de voir quelles sont les modifications nécessaires à réaliser sur les procédures d'un monitor afin qu'elles soient synchronisées correctement, nous allons examiner les différentes actions de synchronisation qui doivent être présentes dans une procédure.

(*) L'implémentation présentée en [ 31 ] ne concernait cependant que des monitors conventionnels sans appels imbriqués (ce qui n'est pas le cas ici !).

D'après le comportement de l'exécution d'une procédure décrit en III.6-7, (comportement qui "intuitivement" garantit une synchronisation correcte de la procédure), nous pouvons identifier 4 actions de synchronisation possibles.

*1. L'action "entrée".*

Cette action se situe en tête d'une procédure et a pour objectif de soumettre l'entrée de la procédure (*) dans le monitor à la condition suivante :

"aucune procédure n'est en cours dans le monitor".

Est-ce cependant suffisant comme condition d'"entrée" ? La réponse est NON ! En effet, il se peut que dans le monitor, aucune procédure ne soit en cours, mais qu'il y ait des procédures qui sont suspendues sur des opérations *wait(a)*, avec *(a:B)* qui est vrai.

Ces procédures ont donc l'occasion de repartir (une seule au maximum pourra repartir), mais elles seront empêchées par l'"entrée" de la nouvelle procédure.

Une procédure *P* suspendue sur un *wait(a)*, avec *(a:B)* qui est vrai un nombre infini de fois peut donc rester bloquée indéfiniment, car chaque fois qu'elle a l'occasion de repartir, une nouvelle procédure "entre" dans le monitor et empêche *P* de repartir.

Nous allons donc soumettre l'"entrée" d'une procédure dans le monitor à la condition suivante : *"aucune procédure n'est en cours dans le monitor* **et** *aucune procédure bloquée sur un wait(a) n'a l'occasion de repartir"*.

Nous dirons dorénavant que le monitor es **OUVERT** lorsque cette condition d'"entrée" est vraie, et **FERME** sinon (N.B. : initialement, le monitor est OUVERT).

L'action "entrée" devra donc réaliser la fonction suivante :

- *attendre que le monitor soit OUVERT, puis signaler que le monitor est FERME* (car une procédure entre en cours).

(*) Lorsqu'on parle d'une procédure en termes dynamiques, il faut comprendre le processus qui exécute cette procédure.

Cette fonction peut facilement être réalisée au moyen de sémaphore. Il suffit pour cela de :

- déclarer dans le monitor un sémaphore *mutex* dont la valeur sera égale à 0 ou 1.

  *mutex* = *1* $\Longleftrightarrow$ le monitor est *OUVERT*.
  (initialement : *mutex* = *1*)

- coder l'action "entrée" au moyen de l'opération *P(mutex)* $\{$ attendre que mutex = 1, puis mettre *mutex* à *0* $\}$.

*2. L'action "sortie".*

Cette action se situe en fin d'une procédure et a pour objectif de signaler que plus aucune procédure n'est en cours.

L'action "sortie" devra donc réaliser la fonction suivante :

Si il existe des procédures bloquées sur des *WAIT*, qui ont l'occasion de repartir

Alors en sélectionner une (par exemple la plus "vieille" : cela permet d'éviter qu'une procédure soit oubliée indéfiniment alors que le nombre d'occasions où elle peut repartir est infini) et lui signaler qu'elle peut repartir.

Sinon signaler que le monitor est *OUVERT* (car aucune procédure n'est en cours et aucune procédure bloquée sur un *WAIT* n'a l'occasion de repartir).

N.B. : avant que l'action "sortie" s'exécute, le monitor est *FERME*, car une opération était en cours.

Comment implémenter cette fonction au moyen de sémaphores ? Nous allons associer à chaque variable **CONDITION** $a_i$ ($1 \leq i \leq c$, $c$ étant le nombre de variables *CONDITION* du monitor, et $a_i$ désigne la $i^\circ$ variable *CONDITION*) :

*) un sémaphore *s(i)*, qui permettra de suspendre des procédures sur des *wait(ai)* et de réveiller des procédures bloquées sur des *wait(ai)*.

*"s(i) = 1"* signalera qu'une procédure bloquée sur un *wait(ai)* peut repartir.

On peut donc dire que $\forall\ 1 \leq i \leq c$,

- à tout instant, un seul $s(i)$ peut être égal à 1, car une seule procédure bloquée sur un *wait* peut réellement repartir.
- $s(i)$ pourra être égal à 1 uniquement lorsque aucune procédure n'est en cours et des procédures sont bloquées sur des *wait(ai)*, avec *(ai:B)* qui est vrai.

N.B. : - $mutex = 0 \Leftrightarrow \exists\ 1$ *et* $1$ *seul* $i$ *tq* $s(i) = 1$
(cela lorsqu'aucune procédure n'est en cours).
- initialement : $\forall\ 1 \leq i \leq c,\ s(i) = 0$

*) une variable entière *count(i)*, qui indiquera le nombre de procédures bloquées sur *ai*.
Initialement : $\forall\ 1 \leq i \leq c,\ count(i) = 0$.

Nous pouvons maintenant facilement implémenter la fonction "sortie" : dénommons par $E$ l'ensemble des indices des variables CONDITION sur lesquelles sont bloquées des procédures qui ont l'occasion de repartir.

$$E = \left\{ i \text{ tq } - 1 \leq i \leq c \right.$$
$$\left. - count(i) > 0 \text{ et } (ai:B) \text{ est vrai} \right\}$$

L'instruction si - alors - sinon de la fonction "sortie" est alors codée de la façon suivante :

Si $E$ est non vide
alors -sélectionner un élément de $E$, soit $i$ (*)
-$V(s(i))$ { signale qu'une procédure bloquée sur un *wait(ai)* peut repartir }
sinon $V(mutex)$ ;

N.B. - $E$ est déterminé juste avant l'instruction si - alors - sinon.
- avant l'action "sortie", une opération est en cours
$\Rightarrow mutex = 0$ et $\forall\ 1 \leq i \leq c : s(i) = 0$

(*) On suppose que l'élément sélectionné est l'indice de la variable CONDITION sur laquelle est bloquée la plus vieille procédure parmi celles qui ont l'occasion de repartir !

### *3. L'action "wait(ai)".*

Cette action se situe lors d'une opération *wait(ai)* (*ai* étant la *i°* variable CONDITION du monitor) et a pour objectif de suspendre la procédure tant que *(ai:B)* est faux, puis de reprendre l'exécution.

N.B. : si l'exécution de la procédure doit être suspendue, on considère que la procédure n'est plus en cours, ce qui permet à une autre procédure soit d'"entrer" dans le monitor, soit de reprendre une exécution qui a été suspendue lors d'un *wait*.

L'action *wait(ai)* doit donc réaliser la fonction suivante :

Si (ai:B) est faux

alors a) signaler qu'une procédure de plus sera bloquée sur un *wait(ai)*.

b) signaler que plus aucune procédure n'est en cours (car la seule procédure en cours va se suspendre).

c) attendre que (ai:B) soit vrai et qu'aucune procédure ne soit en cours, puis signaler qu'une procédure est en cours.

d) signaler qu'une procédure de moins est bloquée sur un *wait(ai)*.

sinon ne rien faire !

Nous pouvons facilement coder les 4 points (a)-d)) de la clause "alors" :

a) étant donné la définition de la variable *count(i)*, il suffit d'incrémenter de 1 cette variable.

*count(i) := count(i)+1 ;*

b) ce point doit signaler que plus aucune opération n'est en cours : son objectif est donc identique à celui d'une action "sortie" $\Rightarrow$ le code de b) est exactement le même que le code d'une action "sortie".

c) ce point est codé au moyen d'un *P(s(i))* : on va attendre que "*s(i) = 1*" (c-à-d qu'aucune procédure ne soit en cours et que (ai:B) soit vrai), puis on mettra *s(i)* à 0 car il y a une procédure en cours.

N.B. - rappelons que *s(i)* est mis à 1 lors d'une action "sortie".

- *s(i) = 1* $\Rightarrow$ *mutex = 0* et *s(j) = 0*, $\forall$ *j* $\neq$ *i* : cela signifie que lorsque "*s(i) = 1*", seule une procédure bloquée sur un *P(s(i))* - et il y en a au moins une - peut repartir.

d) *count(i) := count(i)-1 ;*

N.B. : cette opération aurait très bien pu être exécutée au moment où l'on exécute un *V(s(i))*, c-à-d dans une action "sortie".

*4. L'action "appel".*

Cette action se situe lors d'un appel de procédure et peut être décomposée en deux sous-actions :

- L'action *appel-avant* qui se situe juste avant un appel de procédure et a pour objectif de signaler que plus aucune procédure n'est en cours (car un appel de procédure provoque la sortie de la procédure appelante jusque quand la procédure appelée est terminée).
  Cette action est donc identique à l'action *sortie*.

- L'action *appel-après* qui se situe juste après un appel de procédure et a pour objectif de faire réentrer la procédure dans le monitor.
  Cette action est donc identique à l'action *entrée*.

*Remarques* : nous avons donc en fait réellement trois formes d'actions de synchronisation :

. L'action *entrée* qui se situe à l'entrée d'une procédure et juste après chaque appel de procédure.

. L'action *sortie* qui se situe à la fin d'une procédure, juste avant chaque appel de procédure, et dans une action *wait(ai)* (au point b) de la clause "alors").
. L'action *wait(ai)* qui se situe lors d'une opération *wait(ai)*.

*5.1.2. ANNEXES*.

La justification et le codage PASCAL des modifications décrites précédemment, accompagnés d'un exemple se trouveront en annexe.

5.1.2.1. Justifications des modifications (en annexe A.I.1).

5.1.2.2. Codage PASCAL des modifications (en annexe A.I.2).

5.1.2.3. Exemple (en annexe A.I.3).

*5.1.3. REMARQUES*.

Nous allons préciser certaines notions relatives à l'implémentation d'un monitor conventionnel. D'abord, nous verrons que dans certains cas particuliers, l'implémentation peut être très simplifiée. Ensuite, nous nous interesserons à l'implémentation d'opération *wait* faisant intervenir des variables locales à une procédure et / ou des notions de priorités.

5.1.3.1. Simplification de l'implémentation.

Nous allons montrer que dans certains cas particuliers, il est possible de simplifier ou même d'éliminer certaines actions de synchronisation dans une procédure.

*1° cas : combinaison "entrée-sortie"*.

Dans une procédure, le code d'une action *entrée* est suivi directement du code d'une action *sortie*, c-à-d :

.
.
.
*"entrée"* ;
*"sortie"* ;
.
.
.

Cela signifie qu'une procédure qui n'était pas en cours dans le monitor, demande à y entrer pour en ressortir tout de suite. L'entrée au cours de cette procédure ne sert donc à rien et va entraîner des suspensions inutiles pour elle-même et pour les autres procédures.

Supprimer cette double action (*entrée-sortie*) permet donc une exécution plus efficace des procédures sans modifier pour autant leurs spécifications.

Nous pouvons donc établir la règle de réduction suivante : lorsqu'une action *entrée* est suivie directement d'une action *sortie*, ces deux actions seront supprimées.

Cette règle sera appliquée lors des 4 situations suivantes (c-à-d les 4 situations où une action *entrée* est suivie directement d'une action *sortie*) :

a) le début d'une procédure est suivi directement de la fin de la procédure (la procédure est donc vide !).

b) le début d'une procédure est suivi directement de l'appel d'une procédure.

c) un appel de procédure est suivi directement de la fin d'une procédure.

d) un appel de procédure est suivi directement d'un autre appel de procédure.

L'application de la règle dans la dernière situation est très interessante car elle permettra d'implémenter une suite consécutive

d'appels de procédure de la manière suivante : une action *sortie* précédera le premier appel et une action *entrée* suivra le dernier appel. Tous les couples *entrée-sortie* entre deux appels ont été supprimés.

N.B. : les autres combinaisons entre des actions *entrée* et des actions *sortie* sont impossibles (*entrée-entrée*, *sortie - sortie* et *sortie-entrée*) dans une procédure, étant donné la structure d'une procédure. En effet, une action *entrée* (*sortie*) se trouve soit en début (en fin) d'une procédure, soit juste après (avant) un appel de procédure : il est donc impossible qu'une autre action de la même procédure la précède (la suive).

*2° cas : combinaison "entrée-wait".*

Dans une procédure, le code d'une action *entrée* est suivi directement d'une action *wait*, c-à-d :

```
                "P(mutex) ;
                si (ai:B) est faux
                alors     count(i) := count(i)+1 ;
                          Déterminer E ;
wait(ai)                  si E est non vide
                "sortie"  alors sélectionner un élément de E : soit j ;
                                V(s(j))
                          sinon V(mutex) ;
                          P(s(i)) ;
                          count(i) := count(i)-1 ;"
```

Juste après une action *entrée*, aucune procédure n'a l'occasion de repartir (car le monitor était OUVERT) $\Longrightarrow$ l'ensemble *E* est vide. Cela signifie que lors de l'exécution d'une action *sortie* qui se trouve dans une action *wait* située juste après une *entrée*, l'ensemble *E* sera toujours vide et donc l'opération *V(mutex)* sera toujours exécutée : l'action *sortie* peut donc être réduite à un *V(mutex)*.

Nous pouvons donc établir une deuxième règle de réduction : l'action *sortie* se trouvant dans un *wait* situé juste après une

action *entrée* sera réduite à une opération *V(mutex)*.

N.B. : - cette règle peut être appliquée dans l'exemple donné en annexe (A.I.3)
- par expérience, nous savons que les deux combinaisons que l'on vient de voir sont très fréquentes dans un monitor : une implémentation "intelligente" devra donc utiliser au maximum les deux règles de réduction, ceci afin d'éliminer les suspensions et les évaluations inutiles.

### 5.1.3.2. Opérations "wait" avec variables locales.

Jusqu'à présent, les opérations *wait(ai)* que nous avons vues étaient du type *globales*, c-à-d que *(ai:B)* ne faisait référence qu'à des variables globales du monitor. Nous allons maintenant voir des opérations *wait(ai)* locales, c-à-d où *(ai:B)* pourra dépendre du contexte de la procédure qui exécute l'opération.

*Exemple* : un monitor contient les variables globales *I*, *J* et les variables CONDITION *a* : **COND** [ *(J-I) > 0* ] ;
*b* : **COND** [ *10-(J-I)-lg > 0* ] ;
(*lg* n'est pas une variable globale).

- Les opérations *wait(a)* seront globales.
- Les opérations *wait(b)* seront locales aux procédures qui les exécutent car la valeur de *(10-(J-I)-lg > 0)* dépendra non seulement des variables globales *I,J* mais aussi de la valeur de *lg*, qui doit être une variable locale à la procédure qui exécute l'opération.

N.B. : l'implémentation devra vérifier que :
- les procédures qui utilisent les opérations *wait(b)* contiennent une variable locale (qui peut être un paramètre) de nom *lg*.
- les procédures qui utilisent *wait(a)* et / ou *wait(b)* ne contiennent pas de variables locales de nom *I* ou *J*, car *I* et *J* doivent faire référence aux variables globales.

Lorsqu'une procédure exécutée par un processus *Pk* arrivera sur une opération *wait(ai)* locale, avec *(ai:B)* qui est faux dans le contexte de *Pk* (le contexte de *Pk* se limite ici au contexte de la procédure exécutée par *Pk*), il ne faut plus que cette procédure aille se suspendre sur un sémaphore associé à *ai* (un sémaphore *global*) comme c'était le cas auparavant mais sur un sémaphore *privé* au processus *Pk*. Cela est du au fait qu'il ne faudra plus signaler que *(ai:B)* est vrai, mais il faudra signaler que "*(ai:B)* est vrai dans le contexte du processus *Pk*".

Nous allons donc supprimer les sémaphores globaux *s(1)...s(x)*. De même, les variables entières *count(1)...count(c)* deviennent inutiles car nous ne devons pas savoir le nombre de processus bloqués sur un *wait(ai)* mais l'identité de ces processus.

A la place, nous allons donc déclarer :

*) les sémaphores privés *t(1)...t(p)* où *p* sera le nombre de processus qui peuvent utiliser le monitor.

$\forall\ 1 \leq k \leq p$ : *t(k)* est le sémaphore privé du processus *Pk*.

$t(k) = 1 \iff$
- aucune procédure n'est en cours.
- une procédure exécutée par *Pk* est bloquée sur un *wait (ai)*.
- *(ai:B)*est vrai dans le contexte de *Pk* (*).
- le processus *Pk* est le plus vieux parmi ceux qui peuvent repartir.

*) les listes *list(1)...list(c)* où *list(i)* est la liste des n° de processus bloqués sur des *wait(ai)*.

N.B. : . $t(k) = 1 \Longrightarrow \exists\ !\ i$ tq - $1 \leq i \leq c$
- $k \in list(i)$

. Initialement : $\forall\ 1 \leq k \leq p$ ; t(k) = 0
$\forall\ 1 \leq i \leq c$ : list(i) = { }

*) Rappelons une dernière fois que ce contexte se limite à celui de la procédure exécutée par *Pk*.

*Nous pouvons maintenant "coder" les actions sortie et wait(ai) :*

*1. Action "sortie"*

Cette action devra réaliser la fonction suivante :

*Si* il existe des processus bloqués sur un WAIT qui ont l'occasion de repartir
*Alors* en sélectionner un (le plus "vieux") et lui signaler qu'il peut repartir ( $\Longrightarrow$ on lui envoie un signal sur un sémaphore privé).
*Sinon* signaler que le monitor est OUVERT.

Si on dénomme par *E* l'ensemble des processus bloqués sur des WAIT qui ont l'occasion de repartir (*E* est donc un ensemble de numéros de processus !), cette fonction peut être implémentée de la manière suivante :

*"Déterminer E ;* $\left\{ E = \left\{ k \text{ tq } - 1 \leq k \leq p \right.\right.$
$\left. - \exists\, i \text{ tq } - 1 \leqslant i \leqslant c \right.$
$- k \in list(i)$
$-$ *(ai:B) est vrai dans le contexte de Pk*
$\left.\right\}$
$\left.\right\}$

*Si* *E est non vide*
*alors* - *sélectionner le plus "vieil" élément de E : soit k ;*
- *V(t(k))*
*sinon* *V(mutex) ;"*

*2. Action "wait(ai)".*

N.B. : on suppose que chaque procédure du monitor contiendra une variable locale *procnum* qui contiendra le numéro du processus qui exécute la procédure. *Procnum* sera créé et initialisé lors de chaque activation de procédure par un processus.

Une action *"wait(ai)"* sera "codée" de la manière suivante :

*"Si (ai:B) est faux*
*alors - list(i) := list(i)+procnum ;*
*- "action" "sortie" ;*
*- P(t(procnum)) ;*
*- list(i) := list(i)-procnum"*

Lorsqu'un processus exécutera une action *wait(ai)*, il réalisera dans l'ordre les opérations suivantes :

- évaluer *(ai:B)*. *((ai:B)* sera évalué dans son contexte !).
- *Si (ai:B)* est faux
  *alors* - le processus signale qu'il va se bloquer sur un *wait(ai)* $\Longrightarrow$ il plaçe son numéro dans *list(i)*.
  - le processus signale qu'il n'est plus en cours.
  - le processus va se suspendre sur son sémaphore privé.
  - lorsqu'il sera débloqué, le processus signalera qu'il n'est plus bloqué sur un *wait(ai)* : $\Longrightarrow$ il enlève son numéro de *list(i)*.

*Remarques* : .dans le code d'une action *"wait(ai)"*, *i* est une constante et *procnum* est le nom d'une variable locale.
.le code d'une action *"entrée"* ne change pas.

### 5.1.3.3. Opérations "wait" avec "priorité".

Nous allons montrer comment implémenter des opérations *"wait"* faisant intervenir des notions de priorités.

La *syntaxe* de cette opération sera la suivante :

**wait**(*ai*) { **with priority** *p* } où *p* est une constante ou une variable locale de type entier positif.

La *sémantique* (*) de cette opération est la suivante : la procédure qui exécute cette opération sera suspendue tant que *(ai:B)* est faux **et / ou** il existe des procédures suspendues sur des *wait(ai)* qui ont des priorités plus grandes que la sienne (c-à-d *p*).

(*) Cette sémantique est empruntée à HOARE [ 9 ]

N.B. : . une procédure pourra donc être suspendue même si *(ai:B)* est vrai : en effet, il suffit qu'il y ait une autre procédure plus prioritaire qu'elle qui soit elle aussi suspendue sur un *wait(ai)*.

. par défaut, la priorité d'une procédure sera égal à 0. Cela signifie qu'une opération *"wait(ai)"* sera équivalente à l'opération *"wait(ai)* **with priority** *0"*.

*Comment implémenter cela ?*

Nous allons garder - les sémaphores privés $t(1)...t(p)$
- les listes $list(1)...list(c)$

et nous allons associer à chaque processus une variable entière de nom *prior* : lorsqu'une procédure exécutée par le processus *Pk* est suspendue sur un *"wait(ai)* **with priority** *p"*, la variable $prior(k)$ contiendra la valeur de *p*. Sinon, cette variable est indéfinie.

$(k \in list(i) \Longleftrightarrow prior(k)$ *est défini*$)$.

Initialement, les $prior(1)...prior(p)$ sont indéfinis.

Nous pouvons maintenant préciser la condition nécessaire et suffisante pour que *"t(k)=1"*, c-à-d pour qu'une procédure exécutée par le processus *Pk* et bloquée sur un *"wait"* puisse repartir :

$t(k) = 1 \Longleftrightarrow$ . aucune procédure n'est en cours.

. $\exists\ 1 \leqslant i \leqslant c$ *tq* $k \in list(i)$ (cela signifie qu'une procédure exécutée par le processus *Pk* est suspendue sur un *"wait(ai)* **with priority** *p"*, et $prior(k) = p$).

. *(ai:B)* est vrai dans le contexte de cette procédure.

. $\forall\ l \in list(i) : prior(k) \geqslant prior(l)$ (cela signifie qu'il n'existe aucune procédure bloquée sur un *"wait(ai)"* qui ait une priorité plus grande que celle de *PK*).

. la procédure exécutée par *Pk* est la plus vieille parmi celles qui peuvent repartir.

*Nous pouvons maintenant "coder" les actions "sortie" et "wait(ai)"*

*1. action "sortie"*

Elle est identique à l'action *"sortie"* décrite en V.12, exepté la définition de *E :*

$$E = \left\{ k \; tq \; - \; 1 \leq k \leq p \right.$$
$$\left. - \exists \; i \; tq \; - \; 1 \leq i \leq c \right.$$
$$\left. - \; k \in list(i) \right.$$
$$\left. - \; (ai:B) \text{ est vrai dans le contexte de } P_k \right.$$
$$\left. - \; \forall \; l \in list(i) \; : \; prior(k) \geq prior(l) \right\}$$

*2. action "wait(ai)"*

Chaque opération *"wait(ai)* **with priority** *p* sera remplacée par une action qui aura la forme suivante :

*Si* *(ai:B) est faux ou* $\exists \; l \in list(i) / tq \; p < prior(l)$
*Alors* - *list(i) := list(i)+procnum ;*
- *prior(procnum) := p ;*
- *"action" "sortie" ;*
- *P(t(procnum)) ;*
- *list(i) := list(i)-procnum ;*

*Remarques :* - la clause *alors* sera exécutée lorsque soit *(ai:B)* est faux, soit il existe une ou plusieurs procédures bloquées sur des *wait(ai) qui ont une priorité plus* grande que *p*.
- le fait d'enlever *procnum* de *list(i)* entraîne l'"effet de bord" suivant : *prior(procnum)* n'a plus aucun sens (est indéfini).
- le lecteur interessé trouvera dans [ 2 ], [ 9 ] des exemples concrets qui font intervenir des *WAIT* avec variables locales et priorité.

## 5.2. IMPLEMENTATION D'UN MONITOR CONCURRENT.

Nous allons présenter cette section en suivant le même plan que celui utilisé pour le monitor conventionnel, c-à-d que nous commencerons par une présentation des modifications à réaliser sur les procédures d'un monitor concurrent pour qu'elles soient synchronisées correctement.

La justification et le codage PASCAL de ces modifications sera présenté en annexes. Un ensemble de règles permettant de simplifier l'implémentation, et une présentation de quelques problèmes posés par le monitor concurrent se trouveront dans un paragraphe *remarques* qui cloturera cette section.

### *5.2.1. MODIFICATIONS A INTRODUIRE DANS LES PROCEDURES.*

Avant de présenter ces modifications, il est utile de rappeler les deux caractéristiques d'un monitor concurrent :

a) à chaque procédure *P* est associée une liste de procédures qui doivent être mutuellement exclusives avec *P*.

N.B. : . nous dénommerons par *proc(k)* la $k^{\circ}$ procédure déclarée dans le monitor.

. *list(k)* sera la liste des procédures qui doivent être mutuellement exclusives avec *proc(k)*, c-à-d : $i \in list(k) \Longleftrightarrow$ *proc(k)* excludes [ ...*proc(i)*... ] .

. $i \in list(k) \Longleftrightarrow k \in list(i)$ (caractère symétrique de la relation d'exclusion mutuelle).

b) on suppose qu'une contrainte d'exclusion mutuelle a été spécifiée (par le programmeur) sur tout couple de procédures pour lesquelles il existe des interférences. Pour éviter toutes interférences, il suffira donc de veiller à ce que 2 procédures pour lesquelles une contrainte d'exclusion mutuelle a été imposée ne soient jamais en cours en même temps.

Comme pour le monitor conventionnel, nous allons définir 4 actions de synchronisation qui ont lieu dans une procédure : les actions *"entrée"*, *"sortie"*, *"wait"* et *"appel"*.

Etant donné que plusieurs procédures pourront être en cours dans le monitor, plusieurs actions de synchronisation pourraient être exécutées simultanément. Or, nous verrons que ces actions

de synchronisation manipulent un ensemble d'informations communes. Il est donc nécessaire d'imposer une contrainte d'exclusion mutuelle sur ces actions de synchronisation : nous utiliserons pour cela un sémaphore *action*, qui aura la signification suivante :

*action = 1* $\Longleftrightarrow$ *aucune action de synchronisation n'est en cours.*
(initialement : *action = 1*).

1. *L'action "entrée-k"*.

N.B. : nous prenons le terme *"entrée-k"* et non *"entrée"* car nous verrons que le code d'une action *"entrée"* dépend de la procédure dans lequel il se trouve.

Cette action se trouve en tête d'une procédure *proc(k)*. Chaque action *"entrée-k"* sera précédée d'une opération *P(action)*, de manière à assurer l'exclusion mutuelle des actions de synchronisation (nous verrons par après que lorsqu'une action de synchronisation commence, aucune procédure bloquée à l'intérieur du monitor n'a l'occasion de repartir : cela nous permet de garantir que l'entrée en cours d'une procédure venant de l'extérieur ne risque pas d'empêcher l'entrée en cours d'une procédure bloquée à l'intérieur du monitor).

*Que doit faire une action "entrée-k" ?*

Cette action doit soumettre l'entrée en cours d'une procédure *proc(k)* à la condition suivante :

$CE(k)$ { $\forall\ i \in list(k)$, *le nombre de procédures proc(i) en cours est égal à 0.*

Ceci permet d'assurer que des procédures seront en cours en même temps uniquement si il n'existe aucune contrainte d'exclusion mutuelle entre elles.

L'action *"entrée-k"* aura donc la forme suivante :

Si *CE(k)* est vrai
alors { la procédure va entrer en cours }
- signaler qu'il y a une procédure proc(k) en plus qui est en cours.

- signaler qu'il n'y a plus d'action de synchronisation en cours.

sinon { la procédure doit attendre un signal indiquant que *CE(k)* est vrai }

- signaler qu'il y a une procédure *proc(k)* en plus qui attend ce type de signal.
- signaler qu'il n'y a plus d'actions de synchronisation en cours.
- attendre ce type de signal.

N.B. : nous verrons plus tard que ce sont les actions *"sortie"* qui débloquent les procédures suspendues dans des actions *"entrée"* ou *"wait"*. Lorsqu'une action *"sortie"* aura débloqué une procédure *proc(k)* suspendue dans une action *"entrée-k"*, cette même action *"sortie"* essayera de débloquer une autre procédure.
Cependant, le débloquage d'une nouvelle procédure ne peut être réalisé qu'après avoir effectué la mise à jour résultant du débloquage précédent (par exemple, le débloquage d'une procédure *proc(k)* suspendue en *entrée-k* entraîne la M.A.J. du nombre de procédures *proc(k)* en cours (+1) et du nombre de procédures *proc(k)* suspendues en *"entrée"* (-1)).

Cette M.A.J. peut être effectuée
soit par la procédure qui est débloquée : après avoir effectué la M.A.J., cette procédure doit alors signaler à l'action "sortie" qu'elle peut passer au débloquage suivant.
soit par l'action "sortie" elle-même : la M.A.J. est alors anticipative.

Pour des raisons de simplicité et d'efficacité, nous choisirons la 2° solution.

Pour réaliser l'action *"entrée-k"*, nous utiliserons les variables suivantes : . les variables entières $n(1)...n(p)$ où $p$ est le nombre de procédures déclarées dans le monitor et $n(k)$ indiquera le nombre de procédures *proc(k)* en cours dans le monitor.

N.B. : - initialement : $n(k) = 0, \forall\ k$

- *CE(k) est vrai* $\Leftrightarrow \forall\ i \in list(k) : n(i)=0$.

. les variables entières *count-mutex(1)...count-mutex(p)* où *count-mutex(k)* indiquera le nombre de procédures de type *proc(k)* qui attendent un signal indiquant que *CE(k)* est vrai.
N.B. : initialement : *count-mutex(k) = 0,* $\forall$ *k.*

. les sémaphores *mutex(1)...mutex(p)* où *mutex(p)* permettra d'attendre et d'envoyer des signaux indiquant que *CE(k)* est vrai.
La condition pour envoyer un tel signal (c-à-d par une opération *V(mutex(k))* sera que *CE(k)* soit vrai et *count-mutex(k) > 0.*
N.B. : initialement : *mutex(k) = 0,* $\forall$ *k.*

*Nous pouvons maintenant coder l'action "entrée-k" :*

*"P(action) ;*
*Si CE(k)*
*alors - n(k) := n(k)+1 ;*
*- V(action)*
*sinon - count-mutex(k) := count-mutex(k)+1 ;*
*- V(action) ;*
*- P(mutex(k)) ;"*

Remarque : si au moment où est exécutée une action *"entrée-k"*, il n'y a aucune autre action de synchronisation en cours, il peut cependant y avoir des procédures en cours. Il n'y aura pourtant aucun risque d'interférences car une action *"entrée-k"* et une procédure en cours manipulent des ensembles de variables disjoints.

2. *L'action "wait(ai)k"*.

N.B. : nous prenons le terme *"wait(ai)k"* et non *"wait(ai)"* car nous verrons que le code d'une action *"wait"* dépend non seulement de la variable condition *ai* mais aussi de la procédure *proc(k)* dans lequel se trouve l'opération *wait(ai)*.

Cette action se situe lors de chaque opération *wait(ai)* dans une procédure *proc(k)*. Son objectif est de suspendre la procédure tant

que *(ai:B)* est faux, puis de reprendre l'exécution. La forme de cette action sera la suivante :

*P(action)* ;
Si *(ai:B)* est vrai
alors *V(action)* { la procédure *proc(k)* reste donc en cours avec *(ai:B)* qui est vrai et, étant donné que les contraintes d'exclusion mutuelle sont respectées, il n'y a aucune procédure pouvant rendre *(ai:B)* faux qui est en cours }

sinon { la procédure va sortir du monitor (elle ne sera plus en cours) et ne pourra y rentrer que lorsque non seulement *(ai:B)* est vrai mais aussi *CE(k)* doit être vrai (respect des contraintes d'exclusion mutuelle) }
La procédure *proc(k)* devra donc réaliser les opérations suivantes :

- signaler qu'une procédure *proc(k)* sort du monitor $\Longrightarrow$ exécuter une action *"sortie-k"* (cfr. V.22)
- signaler qu'il y a une procédure de plus bloquée dans une action *wait(ai)k*, et qui attend le signal pour repartir : ce signal indiquera que (*CE(k)* et *(ai:B)*) est vrai.
- *V(action)* ;
- attendre le signal pour repartir.

N.B. : rappelons que la M.A.J. résultant du débloquage d'une procédure dans un *wait* sera réalisée par l'action *"sortie"* responsable de ce débloquage.

*Quelles sont les variables nécessaires à cette fonction ?*

Pour chaque action *"wait(ai)k"* possible (nous dirons qu'une action *"wait(ai)k"* est possible si la procédure *proc(k)* utilise l'opération *wait(ai)*), nous allons lui associer un numéro identificateur qui sera compris entre 1 et **nbrwait** (**nbrwait** étant le nombre d'actions *wait(ai)k* possibles). Ce numéro, qui est une

constante, nous permettra de référencer facilement les variables associées à une action *wait(ai)k*.

*Exemple* : soit les actions *wait(ai)k* possibles :

*wait(a1)2* ⟷ *1*
*wait(a1)3* ⟷ *2*
*wait(a1)2* ⟷ *3*
*wait(a3)2* ⟷ *4*
*(nbrwait = 4)*

N.B. : on suppose que le monitor contiendra une table de correspondance qui permettra à partir d'un couple *(i,k)* d'accéder à son identificateur (que l'on dénommera *IDik*) et vice-versa.

Les variables déclarées seront alors les suivantes :

*) les variables entières *count(1)...count(nbrwait)*, où *count (IDik)* indiquera le nombre de procédures suspendues dans des actions *wait(ai)k*.
N.B. : initialement : *count(k) = 0*, $\forall$ *k*

*) les sémaphores *s(1)...s(nbrwait)* où *s(IDik)* permettra de suspendre et de faire repartir des procédures dans des actions *wait(ai)k*. Une opération *V(s(IDik))* ne pourra être exécutée que lorsque : 1) *count(IDik)* > *0*
2) *(ai:B)* est vrai
3) *CE(k)* est vrai
N.B. : initialement : *s(k) = 0*, $\forall$ *k*.

*Nous pouvons maintenant coder l'action "wait(ai)k" :*

*"P(action) ;*
*Si (ai:B)*
*alors V(action)*
*sinon - action "sortie-k" ;*
*- count(IDik) := count(IDik)+1 ;*
*- V(action) ;*
*- P(s(IDik)) ;"*

*Remarques* : - dans ce code, *IDik* sera une constante, car sa valeur est connue au moment de l'implémentation.

- étant donné que les actions de synchronisation ne modifient aucunes variables référencées par un *(ai:B)*, l'évaluation de cette expression peut être faite en concurrence avec d'autres actions de synchronisation. On peut donc coder l'action *"wait(ai)k"* de la façon suivante :

*"Si not (ai:B)*
*alors - P(action) ;*
*- action "sortie-k" ;*
*- count(IDik) := count(IDik)+1 ;*
*- V(action) ;*
*- P(s(IDik)) ;"*

3. *L'action "sortie-k"*.

Cette action se situe en fin d'une procédure *proc(k)* ou lors d'une action *"wait(ai)k"*, et doit signaler qu'il y a une procédure *proc(k)* en moins qui est en cours dans le monitor (par l'opération *"n(k) := n(k)-1"*). Ce n'est cependant pas suffisant car l'exécution de cette opération peut donner l'occasion de repartir à des procédures suspendues à l'intérieur du monitor. L'action *"sortie-k"* doit donc débloquer toutes celles qui peuvent repartir.

*Quelles sont les procédures qui peuvent être débloquées ?*

Une procédure *proc(j)* peut être suspendue dans deux actions :

a) elle peut être bloquée dans une action *"entrée-j"* car *CE(j)* était faux ( $\Longrightarrow \exists\, i \in list(j)$ *tq* $n(i) > 0$). *CE(j)* ne pourra donc devenir vrai que lorsque une procédure *proc(i)*, $i \in list(j)$, quitte le monitor, c-à-d lors d'une action *"sortie-i"*.

b) elle peut être bloquée dans une action *"wait(ai)j"* car *CE(j)* ou *(ai:B)* est faux.

b.1) soit *CE(j)* est faux $\Longrightarrow$ seule une action *"sortie-t"* peut rendre *CE(j)* vrai.
$t \in list(j)$

b.2) soit *CE(j)* est vrai et *(ai:B)* est faux $\Longrightarrow$ tant que *CE(j)* reste vrai, *(ai:B)* restera faux car seules des procédures *proc(t)*, $t \in$ *list(j)*, peuvent modifier des variables référencées par *(ai:B)*. Pour que *(ai:B)* devienne vrai, il faut que *CE(j)* devienne faux $\Longrightarrow$ b.1)

<u>*Conclusion*</u> : on constate donc qu'une procédure *proc(j)* suspendue dans le monitor ne recevra l'occasion de repartir que lors d'actions *"sortie-i"*, $i \in$ *list(j)*, c-à-d uniquement lors d'actions *"sortie"* exécutées par des procédures qui sont mutuellement exclusives avec elle.
Lorsqu'une action *"sortie-k"* est exécutée, elle ne peut donc donner l'occasion de repartir qu'à des procédures qui sont mutuellement exclusives avec elle.
Cette règle nous permet d'éviter qu'une action *"sortie"* aille voir si une procédure peut repartir alors que l'on sait à l'avance qu'elle ne le pourra pas.

*Comment implémenter l'action "sortie-k" ?*

Après avoir signalé qu'il y a une procédure *proc(k)* en moins qui est en cours, cette action va débloquer le <u>maximum</u> de procédures de telle façon que lorsqu'elle se termine, il n'y a plus aucune procédure qui a l'occasion de repartir (les procédures encore bloquées à ce moment-là devront attendre la prochaine action *"sortie"* pour recevoir éventuellement l'occasion de repartir).

Les procédures qui pourront être débloquées se trouveront
<u>soit</u> dans des actions *"entrée-j"*, avec 1) $j \in$ *list(k)*
2) *CE(j)* est vrai
3) *count-mutex(j)* $> 0$

<u>soit</u> dans des actions *"wait(ai)j"* avec 1) $j \in$ *list(k)*
2) *count(IDij)* $> 0$
3) *CE(j)* est vrai
4) *(ai:B)* est vrai

Nous allons donc déterminer deux ensembles :

*) L'ensemble ***ENTREE*** qui contiendra les numéros des actions *"entrée"* dans lesquelles sont bloquées des procédures qui peuvent repartir.

$$ENTREE = \left\{ j \text{ tq } \begin{array}{l} - \; j \in list(k) \\ - \; count\text{-}mutex(j) > 0 \\ - \; CE(j) \text{ est vrai} \end{array} \right\}$$

*) L'ensemble ***WAIT*** qui contiendra les identificateurs d'actions *"wait"* dans lesquelles sont bloquées des procédures qui peuvent repartir.

$$WAIT = \left\{ IDij \text{ tq } \begin{array}{l} - \; j \in list(k) \\ - \; count(IDij) > 0 \\ - \; CE(j) \text{ est vrai} \\ - \; (ai{:}B) \text{ est vrai} \end{array} \right\}$$

Si ces deux ensembles sont vides
alors cela signifie qu'il n'y a aucune procédure qui a l'occasion de repartir : l'action *"sortie-k"* est donc terminée.
sinon on sélectionne un élément de *ENTREE U WAIT* : on fera en sorte de choisir l'élément le plus "vieux", c-à-d sur lequel est bloquée la plus vieille procédure parmi celles qui ont l'occasion de repartir.

Soit cet élément appartient à *ENTREE* (soit *j*) : on va débloquer une des procédures bloquées dans une action *"entrée-j"* et effectuer la M.A.J. qui résulte de ce débloquage :

- *V(mutex(j)) ;*
- *n(j) := n(j)+1 ;*
- *count-mutex(j) := count-mutex(j)-1 ;*

Après le débloquage d'une procédure suspendue en *"entrée-j"*, il faut retirer des ensembles *ENTREE* et *WAIT* tous les éléments qui ne répondent plus à la définition de ces ensembles. (N.B. : l'inverse ne se produira jamais, c-à-d que un élément qui ne répondait pas à la définition d'un des ensembles avant le débloquage d'une procédure, ne répondra sûrement pas à la définition de l'ensemble après le débloquage !).

*Quels sont les éléments à retirer ?*

.) *count-mutex(j)* a pu devenir égal à 0, par l'exécution de *"count-mutex(j) := count-mutex(j)-1"*

$\Longrightarrow$ retirer l'élément *j* de *ENTREE* si *count-mutex(j) = 0.*

.) $\forall$ *i* $\in$ *list(j), CE(i)* est faux car *n(j)* > *0*

$\Longrightarrow$. retirer de l'ensemble *ENTREE* tous les éléments *i* (avec *i* $\in$ *list(j)*)

. retirer de l'ensemble *WAIT* tous les éléments *IDmn* (avec *n* $\in$ *list(j)*)

Soit cet élément appartient à *WAIT* (soit *IDij*)

on va débloquer une des procédures bloquées dans une action *wait(ai)j* et effectuer la M.A.J. qui résulte de ce déblocquage :
- *V(s(IDij))* ;
- *n(j) := n(j)+1* ;
- *count(IDij) := count(IDij)-1* ;

Après le débloquage d'une procédure suspendue en *"wait(ai)j"* il faut retirer des ensembles *ENTREE* et *WAIT* tous les éléments qui ne répondent plus à la définition de ces ensembles.

Nous allons donc retirer l'élément *IDij* de *WAIT*, si

*count(IDij) = 0*

- retirer de l'ensemble *ENTREE* tous les éléments *n* (avec *n* $\in$ *list(j)*).
- retirer de l'ensemble *WAIT* tous les éléments *IDmn* (avec *n* $\in$ *list(j)*).

Une fois le débloquage et la M.A.J. effectués, on recommence le cycle "Si ces deux ensembles sont vides
alors ...
sinon ...

*Une action "sortie-k" aura donc la forme suivante :*

*P(action) ; (*)*
*n(k) := n(k)-1 ;*
*"Determiner les ensembles ENTREE et WAIT" ;*
*while (ENTREE U WAIT) <> [ ]*
*do - sélectionner un élément de ENTREE U WAIT*
*- { débloquage + M.A.J }*
*endo ;*
*V(action) ; (*)*

*Remarques* : a) le code PASCAL d'une action *"sortie-k"* sera présenté en annexes (en A.II.2).

b) le nombre de cycles dans une action *"sortie-k"* sera fini car - à chaque cycle, le nombre de procédures suspendues à l'intérieur du monitor est décrémenté de 1.
- une condition nécessaire pour entamer un cycle est qu'il y ait au moins une procédure suspendue dans le monitor.

c) on peut remarquer que lorsqu'on calcule l'ensemble WAIT (cfr V.24), on doit évaluer des expressions *(ai:B)* qui sont référencées par des procédures *proc (j)*. Afin d'être sûr qu'il n'y aura aucune interférence lors de l'évaluation de ces expressions, il faut n'évaluer *(ai:B)* que si *CE(j)* est vrai.

*4. L'action appel-k.*

Cette action survient lors d'un appel de procédure réalisé par une procédure *proc(k)*. Comme dans le monitor conventionnel, cette action est décomposée en deux sous-actions :
- L'action *appel-avant*, équivalente à une action *sortie-k*, et qui se situe juste avant un appel de procédure.
- L'action *appel-après*, équivalente à une action *entrée-k*, et qui se situe juste après un appel de procédure.

(*) si cette action se trouve dans une action *wait(ai)k*, les opérations *P(action)* et *V(action)* doivent être supprimées.

*5.2.2. ANNEXES.*

La justification et le codage PASCAL des modifications décrites précédemment, accompagnés d'un exemple se trouvera en annexe.

5.2.2.1. Justification des modifications (en annexe A.II.1)

5.2.2.2. Codage PASCAL des modifications (en annexe A.II.2)

5.2.2.3. Exemple (en annexe A.II.3.)

*5.2.3. Remarques.*

Comme pour le monitor conventionnel, nous allons terminer cette section par quelques remarques : nous présenterons d'abord un ensemble de "règles de réduction" permettant d'obtenir une implémentation très simplifiée lors de cas particuliers. Ensuite, nous nous interesserons à quelques problèmes posés par l'implémentation d'un monitor concurrent.

5.2.3.1. Simplification de l'implémentation.

Nous allons présenter une liste de cas particuliers pour lesquels l'implémentation peut être très simplifiée.

N.B. : cette liste n'est bien sur pas exhaustive, mais les cas présentés répondent aux critères suivants :

- facilement décelables lors de l'implémentation.
- les règles de réduction qui leur sont associées sont facilement applicables.
- apparaissent fréquemment dans un monitor concurrent.

*1. Combinaison "entrée-k - sortie-k".*

Le code d'une action *entrée-k* est suivi directement du code d'une action *sortie-k*. La règle de réduction associée à ce cas consiste à supprimer ces 2 actions.

Les raisons de cette suppression et les cas où survient cette combinaison sont décrits en V.7-9.

2. *Combinaison "entrée-k - wait(ai)k"*.

Le code d'une action *entrée-k* est suivi directement du code d'une action *wait(ai)k*, c-à-d :

```
" {la procédure n'est pas encore en cours}
      P(action) ;
      if CE(k)
      then begin n[ k ] := n[ k ]+1 ; V(action) end
      else begin count-mutex[ k ] := count-mutex[ k ]+1;
                 V(action) ; P(mutex[ k ])
           end ;
{la procédure est en cours}
      if not (ai:B)
      then begin P(action) ;
                 count[ IDik ] := count[ IDik ]+1 ;
                 "sortie-k" ;
                 V(action) ;
                 P(s[ IDik ])
           end ; "
```

Ces deux actions peuvent être combinées en une seule de la manière suivante :

```
" P(action) ;
  if CE(k) and (ai:B)
  then begin n[ k ] := n[ k ]+1 ; V(action) end
  else begin count[ IDik ] := count[ IDik ]+1 ;
             V(action) ;
             P(s[ IDik ])
       end ; "
```

L'interêt de cette simplification est la disparition de l'action *sortie-k* (qui est très coûteuse !) qui n'a plus de raison d'être car la procédure n'est pas encore entrée en cours.

3. *Cas où il existe un $1 \leq k \leq p$ tq list(k) = { }*.

Le monitor contient une procédure qui n'est mutuellement

exclusive avec aucune autre.

N.B. : nous imposerons qu'une telle procédure ne contienne aucune opération *wait(ai)* car sinon, cela signifierait qu'il doit y avoir une autre procédure qui puisse modifier *(ai:B)* et donc qui serait mutuellement exclusive avec *proc(k)*.

L'implémentation d'une telle procédure est très simple : elle consiste à ne réaliser aucune modification sur cette procédure (ni action *entrée-k*, ni action *sortie-k*), car la procédure ne doit ni signaler qu'elle entre en cours, ni signaler qu'ell quitte le monitor.

Pratiquement, l'implémentation d'un monitor concurrent contenant une procédure *proc(k)*, avec *list(k)* = $\{\ \}$ revient en fait à implémenter le même monitor, mais en faisant "comme si" la procédure *proc(k)* n'avait pas été déclarée dans le monitor (c-à-d que l'on ne s'en occupe pas !).

*Exemple* : supposons que l'on ait 4 procédures *p1*, *p2*, *p3* et *p4* avec *list(3)* = $\{\ \}$.
L'implémentation de ce monitor revient donc à implémenter le même monitor mais avec 3 procédures : *p1*, *p2* et *p4*.
La procédure *p3* sera laissée telle quelle.

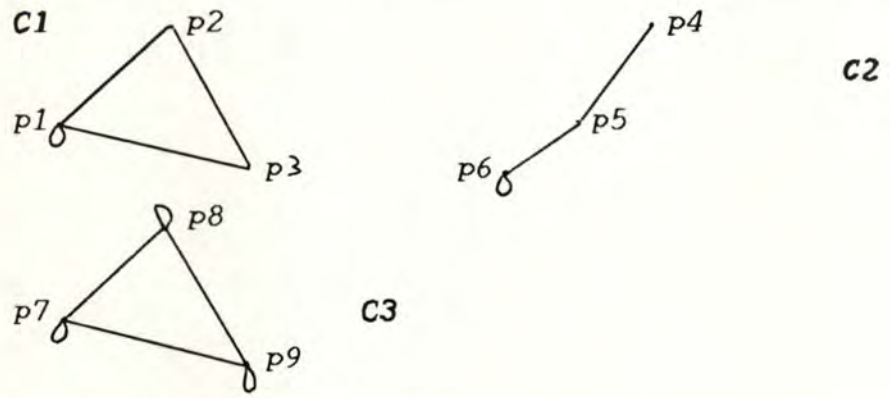
### 4. *Sous-ensembles de procédures indépendants*.

Nous allons expliquer ce cas particulier par un exemple. Supposons que l'on ait un monitor concurrent M avec les procédures *p1*, *p2*,...*p9* et les contraintes d'exclusion mutuelle suivantes :

$$p1 \longleftrightarrow \{p1, p2, p3\} \qquad p2 \longleftrightarrow \{p1, p3\}$$
$$p3 \longleftrightarrow \{p1, p2\} \qquad p4 \longleftrightarrow \{p5\}$$
$$p5 \longleftrightarrow \{p4, p6\} \qquad p6 \longleftrightarrow \{p5, p6\}$$
$$p7 \longleftrightarrow \{p7, p8, p9\} \qquad p8 \longleftrightarrow \{p7, p8, p9\}$$
$$p9 \longleftrightarrow \{p7, p8, p9\}$$

N.B. : les procédures qui ne sont mutuellement exclusives avec aucune autre sont ignorées ! (cfr règle 3).

On peut représenter ces contraintes par le graphe suivant :




Comme on le constate, l'ensemble des procédures est partagé en 3 sous-ensembles (**C1**, **C2**, **C3**) complètement indépendants l'un de l'autre, c-à-d qu'il n'y a pas de contraintes d'exclusion mutuelle entre des procédures appartenant à des sous-ensembles distincts. Cela signifie que l'implémentation de la synchronisation des 9 procédures peut être réalisée en implémentant <u>séparément</u> la synchronisation de chaque sous-ensemble (**C1**, **C2**, **C3**). D'une manière pratique, on peut considérer chaque sous-ensemble comme un monitor "virtuel" (*) qui sera implémenté indépendamment des autres (le monitor *M* contient 3 monitors "virtuels").

Dans l'exemple, on aurait :

- une implémentation pour le "monitor" concurrent **C1**.
- une implémentation pour le "monitor" concurrent **C2**.
- une implémentation pour le "monitor" conventionnel **C3**.
  (les 3 procédures *p7*, *p8*, *p9* seraient donc implémentées selon les principes relatifs au monitor conventionnel).

*Avantage de cette implémentation séparée :*

a) <u>temps d'exécution réduit</u> : les actions de synchronisation seront d'une part plus *simplifiées* (par exemple, les actions de synchronisation utilisées par les procédures *p7*, *p8*, *p9* seront celles d'un monitor conventionnel, qui sont beaucoup plus simples que celles d'un monitor concurrent) et d'autre part pourront être exécutées en *concurrence* si elles sont exécutées par des procédures

(*) monitor virtuel qui peut être soit concurrent (**C1**, **C2**).
soit conventionnel (**C3**).

appartenant à des sous-ensembles distincts. Par exemple, des procédures *P3* et *P4* pourront entrer en même temps dans le monitor *M*, car elles entrent chacunes dans un monitor virtuel différent.

b) occupation de mémoire réduite : prenons par exemple les données nécessaires à assurer le respect des contraintes d'exclusion mutuelle du monitor *M* (cfr. V.29). *p=9*

*) Pour une implémentation normale, les données nécessaires sont les suivantes :
- 1 semaphore *action*
- 9 sémaphores *mutex*
- 9 variables entières *count-mutex*
- 9 variables entières *n*
- un tableau *LIST* de type entier et de dimension *(9*9)* $\Longrightarrow$ 81 variables entières.

total : 10 sémaphores et 99 variables entières qui sont ajoutées aux données de *M*.

*) Pour une implémentation séparée, les données nécessaires sont les suivantes :

**C1** $\longrightarrow$
- 1 sémaphore *action*
- 3 sémaphores *mutex*
- 3 variables entières *count-mutex*
- 3 variables entières *n*
- un tableau *LIST* de dimension *(3*3)* $\Longrightarrow$ 9 variables entières

**C2** $\longrightarrow$ *idem*

**C3** $\longrightarrow$ 1 sémaphore *mutex* (monitor conventionnel).

total : 9 sémaphores et 30 variables entières qui sont ajoutées aux données de *M*.

*Remarque* : d'une manière pratique, on utilisera les 4 règles de réduction que l'on vient de voir dans l'implémentation d'un monitor concurrent *M* de la manière suivante :

a) détecter les procédures *proc(k)* tq *list(k)*$=\{\ \}$ et les "supprimer" de *M* ; Il ne s'agit ici évidemment pas d'une véritable suppression mais simplement d'un mécanisme qui permette de signaler à l'implémentation qu'elle ne doit pas tenir compte de ces procédures (règle 3).

b) décomposer l'ensemble des procédures de *M* (celles qui n'ont pas été "supprimées") en sous-ensembles indépendants (règle 4). La décomposition doit être telle que le nombre de sous-ensembles soit le plus grand possible.

c) pour chaque sous-ensemble C :

soit chaque procédure de C est mutuellement exclusive avec elle-même et avec toutes les autres procédures de C ("monitor" conventionnel) $\Longrightarrow$ les procédures de C seront implémentées selon les principes relatifs à l'implémentation d'un monitor conventionnel et en utilisant les règles de réduction associées à ce type d'implémentation.

soit il existe au moins une procédure de C qui n'est pas mutuellement exclusive avec elle-même et / ou qui n'est pas mutuellement exclusive avec une autre procédure de C $\Longrightarrow$ les procédures de C seront implémentées selon les principes relatifs à l'implémentation d'un monitor concurrent et en utilisant les règles de réduction 1 et 2.

#### 5.2.3.2. Problèmes posés par le monitor concurrent.

Nous avons montré comment réaliser une implémentation correcte et efficace (en utilisant les règles de réduction) des procédures d'un monitor concurrent. Nous allons maintenant terminer ce dernier chapitre en examinant 2 problèmes posés par cette implémentation.

Le premier est du à la complexité des actions de synchronisation d'un monitor concurrent, complexité pouvant dans certaines situations rendre un monitor concurrent moins performant qu'un monitor conventionnel.

Le second problème concerne les suspensions indéfinies qui sont plus difficiles à éviter dans un monitor concurrent.

1. *Baisses de performances dues à la complexité des actions de synchronisation.*

On peut facilement voir que les actions de synchronisation d'un monitor concurrent (et spécialement l'action "sortie") sont plus complexes en temps d'exécution que celles d'un monitor conventionnel.

Cette plus grande complexité, combinée au fait que les actions de synchronisation sont exécutées en exclusion mutuelle, peut parfois dégrader les performances d'un monitor concurrent au point qu'il est préferable d'utiliser la version conventionnelle.

Le cas classique où l'on a cette situation est le suivant : un monitor concurrent, où un haut degré de concurrence est possible, est utilisé dans un environnement tel que ses procédures ne sont jamais en cours en même temps (elles sont donc exécutées comme dans un monitor conventionnel). Etant donné que les procédures dans un monitor concurrent demande un temps d'exécution plus long que dans la version conventionnelle, il est préferable ici d'utiliser cette dernière version.

Nous allons maintenant montrer un exemple de monitor concurrent utilisé dans un environnement où un haut degré de concurrence est atteint, mais qui pourtant se révèle moins efficace que la version conventionnelle utilisée dans le même environnement.

Prenons par exemple un monitor concurrent, représentant une base de donnée élémentaire, sur lequel sont définies 2 procédures : la procédure *read* et la procédure *write*, avec les contraintes d'exclusion mutuelle suivante :

$$Read \longleftrightarrow \{ write \} \text{ et } write \longleftrightarrow \{ read, write \}$$

Plusieurs procédures *read* peuvent donc être en cours simultanément.

N.B. : on suppose que les procédures ne contiennent aucune opération *WAIT* et aucun appel de procédure.

Nous supposerons que le temps moyen

- d'une action *entrée* (*read* ou *write*) est de *2 u* (*)
- d'une action *sortie-read* est de *6 u*
- d'une action *sortie-write* est de *10 u* (nous prenons un temps plus élevé car une action *sortie-write* peut devoir débloquer plusieurs procédures *read*)
- d'une opération *read* est de *5 u*
- d'une opération *write* est de *10 u*.

De plus, nous utiliserons le monitor dans un environnement où *100* procédures *read* et *50* procédures *write* seront activées simultanément.

Si on utilise la version conventionnelle de ce monitor, les temps mis pour exécuter les actions *entrée* et *sortie* sont négligeables car ces actions se réduisent à exécuter respectivement les opérations *P(mutex)* et *V(mutex)* : dans cette version, le temps mis pour exécuter les 150 procédures sera donc approximativement de *1000 u (100*5 + 50*10)*.

Regardons maintenant ce qui se passe avec le monitor concurrent : supposons par exemple que les 100 procédures *read* exécutent d'abord leur action *entrée* (en exclusion mutuelle !).

*) A la *200° u*, on a donc l'état suivant :

- *100* procédures *read* sont en cours dans le monitor et la plupart attendent déjà d'exécuter leur action *sortie*.
- à partir de la *200° u*, les 50 procédures *write* peuvent à leur tour exécuter leur action *entrée* (cela prendra *100 u*).

*) A la *300° u*, on a donc l'état suivant :

- *100* procédures *read* sont en cours dans le monitor et attendent toutes d'exécuter leur action *sortie*.
- *50* procédures *write* sont suspendues dans leur action *entrée* et le resteront tant que des procédures *read* sont en cours.
- à partir de la *300° u*, les *100* procédures *read* vont exécuter leur action *sortie* chacune à leur tour (cela prendra *600 u*).

*) *u* est une unité de temps quelconque !

*) A la *900° u*, on a donc l'état suivant :
- les *100* procédures *read* sont terminées.
- une procédure *write* est en cours : elle vient d'être débloquée par la dernière opération *sortie-read* qui a été exécutée. Cette procédure mettra *10 u* pour exécuter ses opérations, et *10 u* supplémentaires pour exécuter son action *sortie* (et débloquer la prochaine procédure *write*). Il faudra donc *1000 u* pour exécuter les 50 procédures *write*.

*) A la *1900° u*, les 150 procédures auront donc toutes été exécutées.

On constate donc qu'avec cette version où cependant un haut degré de concurrence est atteint (100 procédures *read* sont en cours simultanément entre la 200° et la 300° *u*), le temps nécessaire à l'exécution des 150 procédures est pratiquement le double de celui demandé par la version conventionnelle.

Ces mauvaises performances s'expliquent aisément : seule l'exécution concurrente de procédures peut apporter un gain de performance par rapport à la version conventionnelle (dans notre exemple, seules les procédures *read* peuvent apporter un gain de performance).

Or, si pour ces procédures, le temps qu'elles consacrent à l'exécution des actions de synchronisation (qui sont mutuellement exclusives !) est plus grand ou égal à celui consacré aux opérations "ordinaires", alors l'exécution concurrente de ces procédures prendra au moins le même temps que l'exécution en exclusion mutuelle de leurs opérations ordinaires, c-à-d le type d'exécution implémenté par la version conventionnelle. Il est alors plus efficace d'utiliser cette dernière version.

Dans notre exemple, seule l'exécution concurrente des procédures *read* peut apporter un gain de performance par rapport à la version conventionnelle. Or, le temps que consacre une procédure *read* à ses actions de synchronisation est de *8 u*, contre *5 u* pour ses opérations ordinaires. Aucun gain de performance n'est donc à attendre, car l'exécution concurrente des 100 procédures *read* prendra au moins *800 u*, c-à-d *300* de plus qu'avec la version conventionnelle.

De plus, l'exécution des 50 procédures *write* dans la version

concurrente prendra *600 u* (50*12) de plus qu'avec la version conventionnelle. Ces *600 u* sont dues à l'exécution des actions de synchronisation des procédures *write*. Pour que la version concurrente soit efficace, il aurait donc fallu non seulement que l'exécution des procédures *read* apportent un gain de performance, mais en plus que ce gain de performance soit tel qu'il compense l'"overload" des procédures *write* : le gain de performance aurait donc du être plus grand que *600 u* (dans notre exemple, il est de *-300 u* !).

Pour que l'exécution concurrente de procédures puisse apporter un gain de performance, il est donc nécessaire que le temps que consacrent ces procédures à leurs actions de synchronisation soit plus petit (et si possible beaucoup plus petit) que le temps consacré à leurs opérations ordinaires.

Un exemple où un gain de performance est obtenu est le suivant : reprenons l'exemple précédent, en changeant simplement le temps d'une opération *read* (*100 u* au lieu de 5) et celui d'une opération *write* (*200 u* au lieu de *10*).

Dans la version conventionnelle, le temps mis pour exécuter les 150 procédures sera approximativement de *20.000 u* (100*100 + 50*200).

Dans la version concurrente, si on reprend les mêmes conditions d'exécution que celles du premier exemple (cfr V.34), il faudra seulement *11.400 u* pour exécuter les 150 procédures :

- *200 u* pour que les 100 procédures *read* entrent dans le monitor.
- *100 u* pour que les 50 procédures *write* se suspendent dans leur action "entrée".
- *600 u* pour que les 100 procédures *read* sortent du monitor.
- *10.500 u* pour que les 50 procédures *write* exécutent leurs opérations et sortent du monitor (50*210).

La version concurrente est donc ici très efficace. L'exécution concurrente des 100 procédures *read* prend *800 u* au lieu des *10.000* nécessaires à la version conventionnelle : le gain de performance est donc de *9200 u*, auxquelles il faut soustraire les *600 u* d'"overload" des procédures *write* $\Longrightarrow$ *8600 u*.

On remarquera que plus le temps que consacre une procédure *read* à ses opérations ordinaires est grand par rapport à celui consacré

à ses actions de synchronisation, plus le gain de performance est proche de son maximum, c-à-d un rapport de 100 à 1 entre le temps nécessaire à l'exécution des 100 procédures *read* dans la version conventionnelle et celui nécessaire à la même exécution dans la version concurrente.

Nous avons donc montré que l'efficacité d'un monitor concurrent dépendait entre autres du rapport entre le temps consacré aux actions de synchronisation et celui consacré aux opérations ordinaires : plus ce rapport est faible (cela surtout pour les procédures qui peuvent être concurrentes) et plus les gains de performance apportés par la version concurrente peuvent être importants.

Le programmeur et l'implémentation ont donc tous les deux un rôle à jouer dans l'efficacité d'un monitor :

- le programmeur, en veillant à ce que les procédures des monitors concurrents qu'il a déclarés consacrent un temps important à leurs opérations ordinaires (c'est le cas par exemple pour des procédures contenant des opérations d'entrée-sortie). Lorsque le temps consacré aux opérations ordinaires est négligeable, il est alors préferable que le programmeur utilise la version conventionnelle du monitor.
- l'implémentation, en veillant à ce que les actions de synchronisation prennent le moins de temps possible. Les règles de réduction que nous avons vues sont prévues pour cela. De plus, on peut imaginer des systèmes qui permettent une exécution très rapide des actions de synchronisation (mémoire *cache* contenant les données référencées par les actions de synchronisation, codage hardware de ces actions, ...).

*Remarques* : - les règles de réduction décrites en 5.2.3.1. sont très importantes car elles permettent souvent d'implémenter un monitor concurrent en utilisant l'implémentation d'un monitor conventionnel (dont les actions de synchronisation sont beaucoup moins complexes en temps d'exécution).

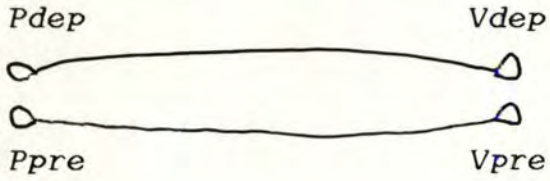
Prenons par exemple les monitors concurrents décrits au chapitre IV pour résoudre les 3 versions du "problème des représentants". Si on les implémente en utilisant les règles de réduction, on constate que :

*) dans les 3 versions, le monitor concurrent *com2000* contient une procédure *ecrire* et cette procédure est mutuellement exclusive avec elle-même $\Rightarrow$ en utilisant les règles de réduction (la règle 4), ce monitor sera décomposé en 1 sous-ensemble qui sera implémenté comme un "monitor" conventionnel.

*) dans la première version, le monitor concurrent *buffer* (cfr IV.8-9) contient 2 procédures et chacune d'elles est mutuellement exclusive avec elle-même et avec l'autre $\Rightarrow$ ce monitor sera aussi implémenté comme un "monitor" conventionnel.

*) dans la deuxième version,. le monitor concurrent *buffer* (cfr IV.12) contient 2 procédures et chacune d'elle est mutuellement exclusive avec elle-même $\Rightarrow$ ce monitor sera décomposé en 2 sous-ensembles indépendants qui seront chacun implémentés comme un "monitor" conventionnel.

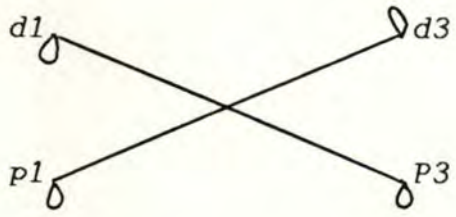
. le monitor concurrent *semaph* (cfr IV.12) contient 4 procédures avec les contraintes d'exclusion mutuelle suivantes :




Ce monitor sera donc décomposé en 2 sous-ensembles indépendants, chacun étant implémenté comme un "monitor" conventionnel.

*) dans la troisième version,. le monitor concurrent *buffer* (cfr IV.14) contient 2 procédures sur lesquelles aucune contrainte d'exclusion mutuelle n'est imposée $\Rightarrow$ les procédures de ce monitor peuvent être laissées telles quelles (règle de réduction n°3). Ce monitor apportera donc des gains de performance très interessants par rapport à la version conventionnelle.

. le monitor concurrent *liste* (cfr IV.14) contient 4 procédures avec les contraintes d'exclusion mutuelle suivantes :




Ce monitor sera donc décomposé en 2 sous-ensembles indépendants, chacun étant implémenté comme un "monitor" conventionnel.

Nous constatons donc que grâce aux règles de réduction 3 et 4, l'implémentation de ces monitors concurrents ne sera jamais plus complexe que celle de leur version conventionnelle. Les problèmes de baisses de performance dues à la complexité des actions de synchronisation d'un monitor concurrent ne se posent donc pas ici : ces monitors concurrents peuvent donc être utilisés sans hésitation..

- nous avons montré un facteur pouvant influencer l'efficacité d'un monitor concurrent. Il en existe encore d'autres et seule une étude approfondie (par simulation par exemple) peut permettre de déterminer ces facteurs et voir comment ils influencent l'efficacité d'un monitor concurrent.

*1. Suspensions indéfinies à l'entrée d'un monitor.*

Nous avons montré en annexes (cfr A.3) que dans un monitor conventionnel correctement synchronisé, une procédure restait bloquée indéfiniment dans une action *entrée* uniquement si il y avait d'autres procédures à l'intérieur du monitor et une au moins de ces procédures avait un temps d'exécution indéfini. Pour éviter les suspensions indéfinies à l'entrée d'un monitor conventionnel, il suffit donc au programmeur de s'assurer que toutes les procédures ont un temps d'exécution fini.

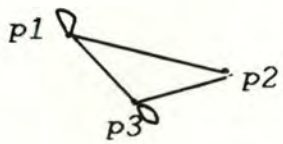
Ce n'est malheureusement plus le cas avec un monitor concurrent : pour éviter qu'une procédure *proc(k)* reste bloquée indéfiniment dans une action *entrée-k*, il faut que *CE(k)* ne reste jamais faux pour toujours (cfr A.11 - 12). Or, nous allons voir que *CE(k)* peut rester indéfiniment faux même lorsque toutes les procédures ont un

temps d'exécution fini. Eviter les suspensions indéfinies à l'entrée d'un monitor concurrent sera donc un problème plus complexe que dans un monitor conventionnel.

*Prenons l'exemple suivant :*

Soit un monitor concurrent avec les procédures *p1, p2, p3* sur lesquelles sont déclarées les contraintes d'exclusion mutuelle suivante :

$$p1 \longleftrightarrow \{ p1, p2, p3 \}$$
$$p2 \longleftrightarrow \{ p1, p3 \}$$
$$p3 \longleftrightarrow \{ p1, p2, p3 \}$$




N.B. : on suppose que les procédures ont toutes un temps d'exécution fini.

Nous allons voir si $\exists\ 1 \leq k \leq 3$ *tq* $CE(k)$ peut rester faux pour toujours, alors que l'on sait qu'aucune procédure ne restera en cours indéfiniment.

Prenons d'abord le cas où $k=2$ : $CE(2)$ sera faux $\Longleftrightarrow$ des procédures de type *p1* et / ou des procédures de type *p3* sont en cours. Or, par les contraintes d'exclusion mutuelle, on sait qu'une seule procédure au plus de type $\{ p1, p3 \}$ peut être en cours à un instant donné, ce qui signifie que $CE(2)$ sera faux $\Longleftrightarrow$ $\exists\ 1\ !$ procédure de type *p1* ou *p3* qui est en cours. Etant donné que les procédures ont un temps d'exécution fini, $CE(2)$ ne restera jamais indéfiniment faux. On peut donc dire avec certitude qu'une procédure *p2* ne restera jamais bloquée indéfiniment à l'entrée du monitor.

Prenons le cas où $K=1$ : $CE(1)$ sera faux $\Longleftrightarrow$ des procédures de type $\{ p1, p2, p3 \}$ sont en cours. Or, par les contraintes d'exclusion mutuelle, on sait que :

*) une seule procédure au plus de type *p1, p3* peut être en cours à un instant donné.

*) une procédure de type *p2* est mutuellement exclusive avec une procédure de type *p1* ou *p3*.

Cela signifie donc que *CE(1)* sera faux $\Longleftrightarrow$

(ou exclusif) $\oplus$ <u>soit</u> 1! procédure *p1* ou *p3* est en cours

<u>soit</u> des procédures *p2* sont en cours

On remarque ici que l'évènement "des procédures *p2* sont en cours" peut rester vrai indéfiniment (et donc *CE(1)* restera faux indéfiniment) même si les procédures *p2* ont un temps d'exécution fini. En effet, il suffit que les procédures *p2* qui <u>sortent</u> du monitor soient chaque fois remplacées par d'autres procédures *p2* qui entrent dans le monitor.

<u>N.B.</u> : pour *k=3*, on fera les mêmes remarques qu'avec *k=1*, c-à-d que *CE(3)* peut rester indéfiniment faux, par l'exécution des procédures *p2*.

On peut donc dire que dans cet exemple, les procédures *p2* peuvent empêcher pendant un temps infini l'entrée en cours de procédures *p1* et / ou *p3*.

D'une manière générale, on dira que dans un monitor concurrent correctement synchronisé, où toutes les procédures ont un temps d'exécution fini, la condition *suffisante* pour qu'une procédure de type *proc(k)* ne reste pas bloquée indéfiniment dans une action *entrée-k* est que - $\forall\ i,j \in list(k) : i \in list(j)$ (et $j \in list(i)$)

- $\forall\ i \in list(k) : i \in list(i)$

c-à-d que *CE(k)* est faux $\Longleftrightarrow \exists$ *1!* procédure de type *proc(i)* qui est est en cours ($i \in list(k)$)

Si on reprend l'exemple ci-dessus, on constate que pour *k=1*, on n'a pas la condition suffisante. En effet, $\{2\} \in list(1)$ mais $\{2\} \notin list(2)$. La propriété "$\forall\ i \in list(1) : i \in list(i)$" est donc fausse.

Idem pour *k=3*. Par contre, pour *k=2*, la condition suffisante est présente car : $\{1\} \in list(2)$ et $\{1\} \in list(1)$

$\{3\} \in list(2)$ et $\{3\} \in list(3)$

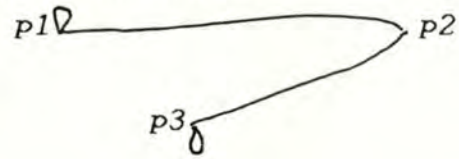
$\{1,3\} \in list(2)$ et $\{1\} \in list(3)$

Prenons maintenant le même exemple, mais avec les contraintes d'exclusion mutuelle qui ont été modifiées :

$p1 \longleftrightarrow \{1,2\}$

$p2 \longleftrightarrow \{1,3\}$

$p3 \longleftrightarrow \{2,3\}$



On remarque que la condition suffisante n'existe toujours pas pour les procédures *p1* et *p3* : cela est du au fait que les procédures *p2* peuvent être concurrentes. Par contre, on constate que pour les procédures *p2*, la condition suffisante qui était vraie auparavant n'existe plus : cela est du au fait que maintenant une procédure *p1* peut être en cours en même temps qu'une procédure *p3*. L'évènement "une procédure *p1* et / ou une procédure *p3* sont en cours" peut rester indéfiniment vrai, même si ces procédures ont un temps d'exécution fini ! Dans cet exemple, il n'y a donc aucune procédure pour laquelle on peut dire avec certitude qu'elle ne sera jamais bloquée indéfiniment dans une action *entrée*.

*Conclusion*

Une manière simple d'éviter les suspensions indéfinies à l'entrée d'un monitor concurrent est donc de construire les monitors en respectant les deux contraintes suivantes :

a) chaque procédure a un temps d'exécution fini.

b) $\forall k$ : - $\forall i,j \in list(k) : i \in list(j)$
- $\forall i \in list(k) : i \in list(i)$

La contrainte b) est cependant trop restrictive car d'une part, elle abaisse fortement le degré de concurrence et d'autre part, un monitor respectant cette contrainte sera en fait un ensemble de "monitor" conventionnel, c-à-d que si on décompose l'ensemble des procédures d'un tel monitor en sous-ensembles indépendants (cfr V.29), on constate que chaque sous-ensemble est un "monitor virtuel" *conventionnel*.

Une autre manière d'éviter les suspensions indéfinies à l'entrée d'un monitor concurrent est de n'imposer que la contrainte a) (qui est donc la même que dans un monitor conventionnel) et de laisser à l'implémentation le soin de veiller à ce que les suspensions indéfinies ne se produisent jamais. Une méthode simple pour réaliser cela est la suivante : désignons par $N$ le nombre de procédures en cours à un instant donné ($N = \sum_{i=1}^{p} n(i)$).

Lorsque $N=0$, on peut affirmer que $CE(k)$ est vrai, $\forall k$. Si on fait en sorte que l'évènement "$N=0$" se produit un nombre infini de fois, on peut donc affirmer que des suspensions indéfinies à

l'entrée du monitor concurrent ne se produiront jamais.

Nous allons donc activer à intervalle régulier une phase de "vidage" dont le rôle consistera à ce que l'évènement "*N=0*" se produise. Cette phase de "vidage" sera réalisée de la manière suivante :

1) condition d'activation : $N > 0$ (la phase de "vidage" n'a de sens que lorsque $N > 0$).
2) condition d'arret : $N=0$ (cette condition se produira lors d'une action *sortie*).
3) pendant la phase de "vidage", aucune procédure ne peut entrer dans le monitor

   $\Longrightarrow$ - une action *entrée-k* réalisée pendant la phase de "vidage" devra provoquer automatiquement la suspension de la procédure, même si *CE(k)* est vrai !

   - une action *sortie* réalisée pendant la phase de "vidage" ne débloquera aucune procédure.

*Remarques* : - on peut renforcer la condition d'activation de la phase "vidage", afin d'éviter des activations inutiles. Par exemple, il est inutile d'activer une phase "vidage" si il n'y a aucune procédure suspendue en *entrée*. De plus, si les seules procédures qui sont suspendues dans des actions *entrée* sont des procédures qui ne risquent pas de suspensions indéfinies (c-à-d des procédures pour lesquelles la condition suffisante décrite en V.41 est présente), alors la phase "vidage" est tout aussi inutile. On peut aussi soumettre l'activation d'une phase de "vidage" à la condition suivante : il faut non seulement qu'il y ait des procédures à risque (c-à-d pour lesquelles des suspensions indéfinies sont possibles) qui soient suspendues dans des actions *entrée*, mais aussi qu'une au moins de ces procédures soit suspendues depuis un temps $\geqslant$ *tps* (*tps* étant une constante indiquant un temps de suspension "suspect").

- on notera que dans les exemples de monitors concurrents décrits au chapitre IV, aucune phase de "vidage" n'est nécessaire car il n'y a aucune procédure "à risque". Par contre, dans l'exemple du monitor concurrent représentant une base de donnée élémentaire (cfr V.33), les phases de "vidage" sont nécessaires. En effet, ce monitor contient les procédures *read* et *write*, avec les contraintes d'exclusion suivantes :

  $read \longleftrightarrow \{write\}$

  $write \longleftrightarrow \{\underline{read}, write\}$

  $\Longrightarrow$ la procédure *write* est une procédure "à risque" (car la condition suffisante décrite en V.41 n'est pas présente pour cette procédure).

Nous avons donc montré dans ce dernier chapitre comment réaliser d'une manière correcte et efficace l'implémentation des deux modèles de monitor : le monitor conventionnel et le monitor concurrent.

Le chapitre suivant, qui clotûrera le mémoire, sera consacré à une conclusion générale sur notre travail.

# CONCLUSION

Nous avons proposé dans ce travail le *monitor concurrent*, version améliorée d'une célèbre primitive de synchronisation de processus coopérants : le *monitor*.

Le monitor fut proposé dans le but de permettre au programmeur de réaliser aisément une synchronisation élégante, ce qui n'est pas le cas avec des primitives plus simples telles que le *sémaphore*.

La résolution d'exemples concrets au moyen du sémaphore et du monitor nous a en effet montré clairement la supériorité de ce dernier dans ce domaine :

- l'implémentation des ressources est cachée aux processus : un changement dans celle-ci peut donc laisser inchangé le code des processus.
- la synchronisation est centralisée : elle est donc plus facile à comprendre et à réaliser.
- un contrôle sur l'accès aux ressources est facilement réalisable.

Au niveau des performances cependant, nous avons pu remarquer la faiblesse du monitor : celui-ci interdit en effet toute exécution simultanée d'opérations sur une même ressource, ce qui n'est pas le cas avec le sémaphore, qui, par sa simplicité laisse au programmeur une totale liberté au niveau des contraintes d'exclusion mutuelle. Nous avons vu, toujours au moyen d'exemples concrets, que cette restriction imposée par le monitor pouvait entraîner de fortes baisses de performances par rapport au sémaphore.

Nous avons alors proposé une nouvelle version du monitor, le monitor concurrent, qui conserve tous les avantages du monitor et qui est en même temps aussi performant que le sémaphore. Pour cela, nous avons gardé toutes les caractéristiques du monitor sauf une : les opérations sur une même ressource ne sont plus automatiquement mutuellement exclusives. C'est au programmeur que revient alors la tâche de définir les contraintes d'exclusion mutuelle sur ces opérations.

Au niveau de l'utilisation, la résolution d'exemples concrets nous a montré que ce monitor concurrent est une primitive aussi

agréable que le monitor, tout en permettant un degré de concurrence identique à celui obtenu avec le sémaphore. Une critique que l'on pourrait cependant faire est le risque d'erreurs que le programmeur court en définissant les contraintes d'exclusion mutuelle sur les opérations.

Au niveau de l'implémentation, réalisée au moyen du sémaphore, le monitor concurrent pose deux problèmes par rapport au monitor conventionnel :

- d'abord, les actions de synchronisation ajoutées par l'implémentation sont plus complexes (en temps et en espace) pour un monitor concurrent. Nous avons présenté des cas où cette plus grande complexité pouvait rendre un monitor concurrent moins performant qu'un monitor conventionnel. Nous avons cependant aussi montré qu'il était souvent possible, d'une part de déceler ces cas par une lecture attentive des opérations du monitor concurrent, et d'autre part, de simplifier fortement les actions de synchronisation d'un monitor concurrent, ce qui limite l'importance de ce problème. Ainsi tous les exemples de monitor concurrent présentés dans ce travail, excepté un, peuvent être implémentés en utilisant l'implémentation d'un monitor conventionnel : le problème présenté ci-dessus ne se pose donc pas pour ces monitors.
- ensuite, le problème des suspensions indéfinies est plus complexe dans un monitor concurrent. Les implémentations que nous avons présentées garantissent au programmeur qu'un processus restera bloqué indéfiniment à l'entrée d'un monitor uniquement si il existe un moment où la condition d'entrée pour ce processus restera fausse pour toujours. Pour un monitor conventionnel, éviter qu'un processus reste bloqué indéfiniment à l'entrée d'un monitor revient alors à s'assurer que les opérations du monitor ont toutes un temps d'exécution fini, ce qui est une tâche relativement facile pour un programmeur.
  Pour un monitor concurrent par contre, nous avons vu que la condition d'entrée d'un processus peut rester indéfiniment fausse, alors que toutes les opérations d'un monitor ont un temps d'exécution fini. Il est alors nécessaire que l'implémentation ajoute dans le monitor un système qui permette

de résoudre ce problème. Nous avons présenté un tel système, caractérisé surtout par sa simplicité : des recherches plus approfondies cependant permettraient certainement de trouver un système plus subtil et plus efficace.

Nous pensons donc que le monitor concurrent présenté dans ce travail apporte un plus interessant par rapport au monitor tout en étant relativement facile à implémenter.

Nous aimerions cependant, pour terminer, donner quelques directions de recherches dont l'aboutissement pourrait donner une suite interessante à notre mémoire.

*) Il y a d'abord le problème des suspensions indéfinies à l'entrée d'un monitor concurrent. Le système que l'on a présenté était très simple : il consiste à "vider" à intervalles réguliers le monitor de toutes opérations en cours, ceci afin qu'à intervalles réguliers, toutes les conditions d'entrée soient vraies. D'autres systèmes, plus subtils et plus efficaces, pourraient être étudiés. On pourrait par exemple augmenter la condition d'entrée de chaque opération d'une condition spéciale autorisant une opération à entrer en cours uniquement si elle ne risque pas de bloquer indéfiniment une autre opération. Un tel système devrait cependant être conçu très prudemment afin d'éviter tout risque d'interbloquage (Dead-Lock).

*) Un autre projet interessant serait d'évaluer par simulation ou par une étude empirique l'efficacité d'un monitor concurrent par rapport au monitor conventionnel. Dans le cas d'une simulation, le choix d'un modèle représentant le monitor serait certainement une des parties les plus délicates du projet.

*) Un troisième projet interessant serait de voir comment implémenter les ressources d'un monitor concurrent afin que ses opérations soient réellement exécutées simultanément.

Ainsi, les N cellules du buffer déclaré dans le monitor concurrent en p. IV.14 nécessitent certainement une implémentation spéciale pour que N opérations puissent y accéder simultanément (une architecture de type *array processor* pourrait par exemple être envisagée).

*) Enfin, un dernier projet serait relatif à la démonstration de la correction des opérations d'un monitor concurrent (*).

On pourrait ainsi voir si les méthodologies qui ont été proposées pour un monitor conventionnel (voir par exemple [31]) peuvent être adaptées à un monitor concurrent, ou si au contraire, il est nécessaire d'en proposer de complètement nouvelles (ce que nous pensons être le cas).

*) Démonstration à ne pas confondre avec la démonstration de la correction des modifications réalisées par l'implémentation.

# ANNEXES

## A.I. Modification des procédures d'un monitor conventionnel.

A.I.1. Justification des modifications.
A.I.2. Codage PASCAL des modifications.
A.I.3. Exemple.

## A.II. Modification des procédures d'un monitor concurrent.

A.II.1. Justification des modifications.
A.II.2. Codage PASCAL des modifications.
A.II.3. Exemple.

# A.I. MODIFICATIONS DES PROCEDURES D'UN MONITOR CONVENTIONNEL.

## *A.I.1. JUSTIFICATION DES MODIFICATIONS.*

Nous allons montrer que les modifications décrites en 5.1.1. assurent une synchronisation correcte des procédures d'un monitor conventionnel. Pour cela, il suffit de montrer que les critères auxquels doivent répondre les procédures pour être synchronisées correctement sont toujours respectés.

Rappelons ces critères (cfr I.28-29) :

1. *Respect des contraintes de synchronisation d'un monitor conventionnel.*

*1.a* : à tout instant, une seule procédure au maximum peut être en cours dans le monitor.

*1.b* : une procédure exécutant une opération *wait(ai)* doit être suspendue tant que *(ai:B)* est faux. Autrement dit, l'exécution d'une opération qui suit directement une opération *wait(ai)* ne peut être réalisée que lorsque *(ai:B)* est vrai !

2. *Pas de suspension inutile.*

Une procédure ne peut être suspendue que si une autre procédure est en cours et / ou elle est sur une opération *wait(ai)*, avec *(ai:B)* qui est faux.

3. *Suspension indéfinie que si...*

Une procédure sera suspendue indéfiniment uniquement si une autre procédure reste en cours indéfiniment dans le monitor ou si elle est sur un *wait(ai)*, avec *(ai:B)* qui est toujours faux lorsque aucune procédure n'est en cours.

*Respect du critère 1.a* : pour montrer le respect de ce critère il suffit de montrer que lorsqu'une procédure "entre en cours" dans le monitor, aucune autre procédure n'est en cours.

Une procédure entre en cours dans le monitor soit par une action "entrée", c-à-d par un *P(mutex)* avec *mutex=1*, soit lors d'un "déblocquage" dans une action *wait(ai)*, c-à-d par un *P(s(i))* avec *s(i)=1*.

Or, on sait que - *mutex=1* $\Longleftrightarrow$ *le monitor est OUVERT*

$\Longleftrightarrow$ - *aucune procédure n'est en cours*

- $\forall$ *1* $\leq$ *i* $\leq$ *c count(i)* $>$ *0*

$\Longrightarrow$ *(ai:B) est faux*

- $\forall$ *1* $\leq$ *i* $\leq$ *c,*
- *s(i)=1* $\Longleftrightarrow$ - *aucune procédure n'est en cours*
  - *count(i)* $>$ *0 et (ai:B) est vrai*
  - *ai est la variable CONDITION sur laquelle est bloquée la plus "vieille" des procédures parmi celles bloquées sur des wait, qui peuvent repartir.*

*Respect du critère 1.b* : on sait que lorsqu'une opération *wait(ai)* (c-à-d une action *wait(ai)*) se termine, *(ai:B)* est vrai et le restera au moins jusqu'à l'exécution de l'opération suivante, car aucune autre procédure n'est en cours.

*Respect du critère 2* : une procédure ne peut être suspendue que sur un *P(mutex)*, avec *mutex=0* ou sur un *P(s(i))*, avec *s(i)=0* et *count(i)* $>$ *0*.

Or, *mutex=0* $\Longrightarrow$ soit une procédure est en cours

soit aucune procédure n'est en cours et il existe un et un seul *i* tq *s(i)=1* (ce qui signifie qu'une procédure bloquée sur un *P(s(i))* est sur le point d'entrer en cours).

*s(i)=0 et count(i)* $>$ *0* $\Longrightarrow$ soit une procédure est en cours

soit aucune procédure n'est en cours et *(ai:B)* est faux.

soit aucune procédure n'est en cours et *(ai:B)* est vrai, mais *ai* n'est pas la variable CONDITION sur laquelle est bloquée la plus vieille des procédures

parmi celles bloquées sur des *wait* qui peuvent repartir (ce qui signifie qu'une autre procédure bloquée sur un *wait (aj)* avec $j \neq i$, est sur le point d'entrer en cours).

*Respect du critère 3* : une procédure ne peut rester bloquée indéfiniment que - sur un *P(s(i))*, c-à-d que *s(i)* reste égal à 0 indéfiniment avec *count(i)* > *0*.
- sur un *P(mutex)*, c-à-d que *mutex* reste égal à 0 indéfiniment.

*) Pour que "*s(i)=0 et count(i)* > *0*" reste vrai indéfiniment, il faut qu'il existe un moment où l'évènement "aucune procédure n'est en cours et *(ai:B)* est vrai" ne se produise plus.
En effet, si cet évènement se répètait indéfiniment, cela signifierait que l'action "sortie" avec "*count(i)* > *0* et *(ai:B)* est vrai" se produirait un nombre infini de fois : l'opération *V(s(i))* serait donc un moment exécutée.
Le critère 3 est donc bien respecté car pour que cet évènement ne se produise plus, il faut soit qu'une procédure reste en cours indéfiniment, soit que chaque fois qu'aucune procédure n'est en cours, *(ai:B)* est faux.

*) Pour que "*mutex=0*" reste vrai indéfiniment, il faut qu'il existe un moment où, pendant que *mutex=0*, l'évènement "une procédure arrive en fin d'exécution" ne se produise plus.
En effet, étant donné que :

1) pendant que *mutex=0*, aucune procédure ne peut entrer dans le monitor.
2) à chaque fois que cet évènement se produit, il y a une procédure en moins dans le monitor,

on peut affirmer que si cet évènement se produisait indéfiniment, il existerait un moment où il n'y aurait plus aucune procédure dans le monitor, ce qui entraînerait l'exécution de *V(mutex)* (car l'action "sortie" serait exécutée avec *E* qui est vide).

Le critère 3 est donc bien respecté car pour que cet évènement ne se produise plus lorsque *mutex=0* (*), il faut

soit qu'une procédure reste indéfiniment en cours.

soit que des procédures dans le monitor (leur nombre est fini car *mutex=0*) exécutent un nombre infini d'action *wait* $\Longrightarrow$ ces procédures ont un temps d'exécution infini, et donc peuvent rester indéfiniment en cours dans le monitor.

### *A.I.2. CODAGE PASCAL DES MODIFICATIONS.*

Les modifications que l'on a présentées en 5.1.1. entrainent des modifications dans les données du monitor (déclaration de nouvelles variables), dans le code d'initialisation du monitor (initialisation de ces nouvelles variables) et dans les procédures.

Nous allons montrer maintenant comment coder en PASCAL ces modifications.

#### *1. Modifications des données du monitor.*

La seule modification à faire est la création des variables suivantes :

- un sémaphore *mutex*.

  mutex : sémaphore ;

- les sémaphores *s(1)...s(c)*, *c* étant le nombre de variables CONDITION du monitor.

  *s : array [ 1..c ] of semaphore ;*

- les variables entières *count(1)...count(c)*

  *count : array [ 1..c ] of integer ;*

N.B. : afin d'éviter des confusions, on supposera que *count* est un mot réservé.

#### *2. Modification du code d'initialisation du monitor.*

Il faut ajouter dans ce code (peu importe l'endroit), les instructions qui initialisent les nouvelles variables du monitor, c-à-d :

*) *mutex=0* $\Longrightarrow$ il y a une procédure en cours dans le monitor

```
" begin var i : integer ;                { i est une variable
      Init(mutex,1) ;                      temporaire }
      for i = 1 to c
      do  begin Init(s[ i ],0) ;
                count[ i ] := 0
          end
end ; "
```

3. *Modification des procédures du monitor*.

Pour chaque procédure du monitor, on réalise les fonctions suivantes :

a) ajouter en tête de la procédure le code d'une action "entrée", c-à-d *"P(mutex) ;"*.

b) ajouter à la fin de la procédure le code d'une action "sortie", c-à-d :

```
" begin var i : integer ;
          E : set of integer ;

      E := [ ] ;
      for i = 1 to c
      do  begin if   count[ i ] > 0 and cond(i)
                then E := E+[ i ]
          end ;
      if E <> [ ]
      then V(s[ select(E) ])
      else V(mutex)
end ; "
```

*Remarque :* ce code utilise deux fonctions que l'on supposera déclarées dans le monitor.
La première de ces fonctions, de nom *select*, reçoit en entrée un ensemble d'indices de variables CONDITION (soit *E* cet ensemble) et renvoie comme valeur le plus "vieil" élément de *E*, celui sur lequel est bloquée la plus vieille procédure parmi celles bloquées sur des éléments de *E*.

La seconde fonction, de nom *cond*, reçoit en entrée un indice de variable CONDITION (soit *i* cet indice) et renvoie la valeur TRUE ssi *(ai:B)* est vrai. Cette fonction sera implémentée de la façon suivante :

```
function COND(i:integer) : boolean ;
   begin COND := false ;
         CASE of i :
              begin [1] ⟶ if (a1:B) then COND := true
                    [2] ⟶ if (a2:B) then COND := true
                     .
                     .
                     .
                    [c] ⟶ if (ac:B) then COND := true
              end
   end ;
```

c) remplacer toutes opérations *wait(ai)* par le code d'une action *"wait(ai)"*, c-à-d :

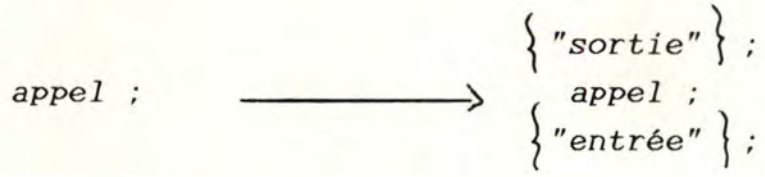
```
" if   not (ai:B)
  then begin count[ i ] := count[ i ]+1 ;
             "code d'une action "sortie"" ;
             P(s[ i ]) ;
             count[ i ] := count[ i ]-1
       end ; "
```

*Remarques :* - dans ce code, *i* est une constante : sa valeur est connue au moment de l'implémentation.

- lorsque les procédures du monitor ont été modifiées et que la fonction *COND* a été déclarée, on n'a plus besoin des variables CONDITION du monitor, ni des expressions booléennes qui leur sont associées : on peut donc les supprimer des données du monitor.

d) ajouter devant chaque appel de procédure le code d'une action "sortie" ;

après chaque appel de procédure le code d'une action "entrée" ;




A.I.3. *EXEMPLE*.

Nous allons montrer comment se présente les données, le code d'initialisation et les procédures d'un monitor conventionnel, lorsque les modifications y ont été appliquées.

Prenons le monitor décrit en III.11 : on remarque qu'il y a deux variables CONDITION (*c=2*) et *a1* réfèrera la première variable (*OKdeposer*), *a2* réfèrera la deuxième (*OKprendre*).

*(a1:B)* désigne l'expression *J-I < N*.

*(a2:B)* désigne l'expression *J-I > 0*.

1) *Les données* :

```
buffer : array[ 0..N-1 ] of com ;
I,J : integer ;
mutex : semaphore ;
s : array[ 1..2 ] of semaphore ;
count : array[ 1..2 ] of integer ;
```

N.B. : rappelons que 2 variables appartenant à 2 monitors distincts peuvent avoir le même nom : c'est l'implémentation qui devra faire la distinction entre ces variables, par exemple en concatenant le nom de la variable avec le code identifiant de son monitor.

2) *Le code d'initialisation* :

```
I := 0 ; J := 0 ;
begin var i : integer ;
      Init(mutex,1) ;
      for i := 1 to 2
      do  begin Init(s[i],0) ;
                count[i] := 0
          end
end ;
```

3) La procédure *deposer (in x:com)* ;

```
begin   P(mutex) ;                {"entrée"}
        if   not (J-I< N)
        then begin count[ 1 ] := count[ 1 ]+1 ;
                   begin var i : integer ;
                         E : set of integer ;

                     E := [ ] ;
                     for i := 1 to 2
"wait(a1)"    "sortie"   do begin if   count[ i ]> 0 and cond(i)
                                 then E := E+[ i ]
                          end ;
                     if  E <> [ ]
                     then V(s[ select(E) ])
                     else V(mutex)
                   end ;
                   P(s[ 1 ]) ;
                   count[ 1 ] := count[ 1 ]-1
             end ;
        buffer[ J mod N ] := x ;
        J := J+1 ;
        begin var i : integer ;
              E : set of integer ;

          E := [ ] ;
"sortie"  for i := 1 to 2
          do  begin if count [ i ]> 0 and cond(i)
                    then E := E+[ i ]
              end ;
          if E ≠ [ ]
          then V(s [select(E)] )
          else V(mutex)
        end
end ;
```

N.B. : on appliquera le même principe pour la procédure *prendre*.
.dans la section *Remarques* (en V.1.3.), on trouvera des règles de réduction qui permettent de simplifier cette implémentation.

## A.II. MODIFICATIONS DES PROCEDURES D'UN MONITOR CONCURRENT.

### *A.II.1. JUSTIFICATION DES MODIFICATIONS.*

Nous allons montrer que les modifications décrites en 5.2.1. assurent une synchronisation correcte des procédures d'un monitor concurrent. Pour cela, il suffit de montrer que les critères, auxquels doivent répondre les procédures pour être synchronisées correctement, sont toujours respectés.

Rappelons ces critères (cfr I.28-29) :

1. *Respect des contraintes de synchronisation d'un monitor concurrent.*

*1.a* : lorsqu'une procédure est en cours dans le monitor, aucune autre procédure avec laquelle elle est mutuellement exclusive ne peut être en cours.

*1.b* : une procédure exécutant une opération *wait(ai)* doit être suspendue tant que *(ai:B)* est faux. Autrement dit, lorsqu'une procédure a terminé une opération *wait(ai)*, *(ai:B)* doit être vrai (et de plus, aucune autre procédure avec laquelle elle est mutuellement exclusive ne peut être en cours).

2. *Pas de suspension inutile.*

Une procédure ne peut être suspendue que si d'autres procédures avec lesquelles elle est mutuellement exclusive sont en cours et / ou elle est sur une opération *wait(ai)*, avec *(ai:B)* qui est faux.

3. *Suspension indéfinie que si...*

Une procédure sera suspendue indéfiniment uniquement si il existe un moment à partir duquel les procédures avec lesquelles elle est mutuellement exclusive seront continuellement en cours, ou bien la procédure est sur une opération *wait(ai)*, avec *(ai:B)* qui sera toujours faux lorsqu'aucune procédure avec qui elle est mutuellement exclusive n'est en cours.

*Respect du critère 1.a* : pour montrer le respect de ce critère il suffit de montrer que :

p1) lorsqu'une procédure entre en cours, il n'y a aucune procédure avec laquelle elle est mutuellement exclusive qui est en cours.

p2) lorsqu'une procédure est en cours, il n'y a aucune procédure avec laquelle elle est mutuellement exclusive qui peut entrer en cours.

Or, on sait qu'une condition nécessaire pour qu'une procédure *proc(k)* entre en cours (lors d'une action *"entrée-k"* ou lors de son débloquage dans une action *"wait(ai)k"*) est que *CE(k)* soit vrai, c-à-d : $\forall\ j \in list(k) : n(j)=0$ . ($\Longrightarrow$ p1) est vrai !).

De plus, on sait que lorsqu'une procédure *proc(k)* a reçu la "permission" d'entrer en cours, $n(k) > 0$

$\Longrightarrow \forall\ j \in list(k)$ : *CE(j) est faux*

$\Longrightarrow$ aucune procédure *proc(j)* ($\forall\ j \in list(k)$) ne peut entrer en cours

$\Longrightarrow$ p2) est vrai.

*Respect du critère 1.b* : nous avons montré lors de la description d'une action *wait(ai)k* (cfr V.19-20) que *(ai:B)* est vrai et qu'il n'y a aucune procédure mutuellement exclusive avec *proc(k)* qui est en cours, lorsque se termine cette action.

*Respect du critère 2* : une procédure *proc(k)* peut être suspendue soit dans une action *"entrée-k"*, sur l'opération *P(mutex(k))* avec *count-mutex(k)* $> 0$ et *mutex(k)*$=0$.

soit dans une action *wait(ai)k*, sur l'opération *P(s(IDik))* avec *count(IDik)* $> 0$ et $s(IDik) = 0$.

N.B. : rappelons qu'une procédure peut être aussi suspendue sur un *P(action)*, ceci afin d'assurer l'exclusion mutuelle des actions de synchronisation.

1°cas : nous allons montrer que si une procédure *proc(k)* est suspendue dans une action *"entrée-k"*, alors *CE(k)* est faux,

c-à-d qu'il y a des procédures avec lesquelles elle est mutuellement exclusive qui sont en cours (ceci ne sera vrai que lorsqu'aucune action *sortie* n'est en cours, car pendant l'exécution d'une telle action, il peut y avoir de "fausses" suspensions, c-à-d des procédures qui sont suspendues mais qui vont être débloquées dans un délai imminent !).

Montrons cela par l'absurde : soit *CE(k)* est vrai (et *count-mutex(k)* > *0*).

Etant donné que *CE(k)* était faux au moment où la procédure s'est suspendue, au moins une action *"sortie-i" (i ∈ list(k))* a du être exécutée, car seul ces actions peuvent faire passer *CE(k)* de faux à vrai. Par hypothèse, la dernière de ces actions a du se terminer avec "*CE(k)* est vrai et *count-mutex(k)* > *0*" (et $k \in list(i)$) $\Longrightarrow$ la dernière action *"sortie-i" (i ∈ list(k))* a du se terminer avec un ensemble *ENTREE* (cfr V.24) non vide, c-à-d contenant l'élément *k*, ce qui est absurde !

2° cas : nous allons montrer que si une procédure *proc(k)* est suspendue dans une action *"wait(ai)k"*, alors *CE(k)* est faux et / ou *(ai:B)* est faux.

La démonstration est similaire à celle du premier cas excepté que - l'hypothèse de départ est : *(CE(k) et (ai:B))* est vrai et *count(IDik)* > *0*.
- on raisonne sur l'ensemble *WAIT* au lieu de l'ensemble *ENTREE*.

*Respect du critère 3* : une procédure *proc(k)* peut être suspendue indéfiniment soit dans une action *"entrée-k"*, sur l'opération *P(mutex(k))* avec *count-mutex(k)* > *0*.
soit dans une action *"wait(ai)k"*, sur l'opération *P(s(IDik))* avec *count(IDik)* > *0*.

N.B. : une procédure ne sera jamais suspendue indéfiniment sur une opération *P(action)*, car les actions de synchronisation ont un temps d'exécution fini !

1° cas : nous allons montrer que si une procédure *proc(k)* est suspendue indéfiniment dans une action *"entrée-k"*, alors cela signifie qu'il existe un moment où *CE(k)* ne sera plus jamais vrai (c-à-d que des procédures avec lesquelles elle est mutuellement exclusive seront continuellement en cours). En effet, si *CE(k)* était vrai un nombre infini de fois, cela voudrait dire qu'un nombre infini d'actions *"sortie-i"*, avec $i \in list(k)$, seraient exécutées avec l'état suivant :

*CE(k) est vrai ; count-mutex(k)* $> 0$ ; $k \in list(i)$

$\Longrightarrow$ un nombre infini d'actions *"sortie-i"* ($i \in list(k)$) seraient exécutées, avec $k \in ENTREE$.

$\Longrightarrow$ il existe un moment où l'élément *k* serait sélectionné, ce qui entrainerait l'exécution de l'opération *V(mutex(k))*.

2° cas : nous allons montrer que si une procédure *proc(k)* est suspendue indéfiniment dans une action *"wait(ai)k"*, alors cela signifie qu'il existe un moment où *"CE(k) et (ai:B)"* ne sera plus jamais vrai (c-à-d que soit *CE(k)* ne sera plus jamais vrai, soit *(ai:B)* sera faux chaque fois que *CE(k)* est vrai). En effet, si *"CE(k) et (ai:B)"* était vrai un nombre infini de fois, cela voudrait dire qu'un nombre infini d'actions *"sortie-i"*, avec $i \in list(k)$, seraient exécutées avec l'état suivant :

*CE(k) est vrai ; (ai:B) est vrai ; count(IDik)* $> 0$ et $k \in list(i)$

$\Longrightarrow$ un nombre infini d'actions *"sortie-i"* ($i \in list(k)$) seraient exécutées avec $IDik \in WAIT$.

$\Longrightarrow$ il existe un moment où l'élément *IDik* serait sélectionné, ce qui entrainerait l'exécution de l'opération *V(s(IDik))*.

### A.II.2. CODAGE PASCAL DES MODIFICATIONS.

Les modifications que l'on a présentées en 5.2.1. entrainent des modifications dans les données du monitor (déclaration de nouvelles variables), dans le code d'initialisation du monitor (initialisation de ces nouvelles variables) et dans les procédures.

Nous allons montrer maintenant comment coder en PASCAL ces modifications.

1. *Modification des données du monitor*.

Les modifications à réaliser dans les données du monitor sont la création des variables suivantes :

+ un sémaphore *action*.

```
action : semaphore ;
```

+ les sémaphores *mutex(1)...mutex(p)*, *p* étant le nombre de procédures déclarées dans le monitor.

```
mutex : array [ 1..p ] of semaphore ;
```

+ les sémaphores *s(1)...s(nbrwait)*, *nbrwait* étant le nombre d'actions *wait(ai)k* possibles.

```
s : array [ 1..nbrwait ] of semaphore ;
```

+ les variables entières *count-mutex(1)...count-mutex(p)* et *count(1)...count(nbrwait)*.

```
count-mutex : array [ 1..p ] of integer ;
      count : array [ 1..nbrwait ] of integer ;
```

+ les variables entières *n(1)...n(p)*.

```
n : array [ 1..p ] of integer ;
```

+ un tableau *LIST* à 2 dimensions.

```
LIST : array [ 1..p ] of array [ 1..p ] of integer ;
```

Ce tableau permettra de réprésenter les listes *list(1)... list(p)* de la manière suivante :

$i \in list(k) \Longleftrightarrow LIST(k,i)=1$

(N.B. : $LIST(k,i) = LIST(i,k)$).

+ un tableau *TABCORR* à 2 dimensions.

```
TABCORR : array [ 1..nbrwait ] of array [ 1..2 ]
of integer ;
```

Ce tableau permettra de représenter les associations entre les actions *wait* et leur identificateur de la manière

suivante : une action *wait(ai)j* a *t* comme identificateur
(*t* représente donc *IDij*)
$\Leftrightarrow$ - $1 \leq t \leq nbrwait$
- $TABCORR[t,1] = i$ et $TABCORR[t,2] = j$

N.B. : - on supposera que *TABCORR* est tel que chaque action *wait(ai)j* possible a un et un seul identificateur.
- les deux dernières variables (*LIST* et *TABCORR*) sont statiques, c-à-d que leur contenu ne change pas au cours de l'utilisation du monitor.

2. *Modification du code d'initialisation du monitor*.

Il faut ajouter dans ce code, les instructions qui initialisent les nouvelles variables du monitor, c-à-d :

```
"begin var i : integer ;
      Init(action,1) ;
      for i = 1 to p
      do begin Init(mutex[i] ,0) ;
               count-mutex[i] := 0 ;
               n[i] := 0
         end ;
      for i = 1 to nbrwait
      do  begin Init(s[i] ,0) ;
                count[i] := 0
          end
end ;"
```

Il reste à produire le code qui initialise les tableaux *LIST* et *TABCORR*. On supposera qu'au moment de l'implémentation, on dispose
- d'un tableau *list* de même type que *LIST* (et de même signification).
- d'un tableau *tabcorr* de même type que *TABCORR* (et de même signification).

De plus, on supposera que ces deux tableaux ont été garnis correctement (par exemple, on a pu les garnir lors de l'analyse syntaxique du monitor).

L'implémentation va donc produire le code suivant :

*"LIST := "list" ; TABCORR := "tabcorr" ;"*

N.B. : dans le code qui aura été produit, on ne retrouvera plus le nom du tableau *list* (ou *tabcorr*) mais le contenu de ce tableau (par exemple, la liste des éléments, lignes par lignes).

3. *Modification des procédures du monitor*.

Une procédure *proc(k)* ($1 \leq k \leq p$) sera modifiée de la manière suivante :

a) ajouter en tête de la procédure le code d'une action *"entrée-k"*, c-à-d : *"P(action) ;*

```
if   CE(k)
then begin n[k] := n[k]+1 ;
           V(action)
     end
else begin count-mutex[k] := count-mutex[k]+1 ;
           V(action) ;
           P(mutex[k])
     end ;"
```

N.B. : nous supposerons que **CE** est une fonction de type booléenne qui a été déclarée dans le monitor. Cette fonction reçoit une valeur entière *k* ($1 \leq k \leq p$) et renvoie la valeur TRUE ssi $\forall\, i \in list(k) : n(i) = 0$, c-à-d :

$$CE(k) = true \Longleftrightarrow \forall\ 1 \leq i \leq p : LIST(k,i) = \mathbf{I} \Rightarrow n(i) = 0$$

b) remplaçer toute opération *wait(ai)* par le code d'une action *wait(ai)k*, c-à-d :

```
"if   not(ai:B)
 then begin P(action) ;
            "code d'une action "sortie-k"" ;
            count[IDik] := count[IDik]+1 ;
            V(action) ;
            P(s[IDik])
      end ;"
```

N.B. : *IDik* est une constante dans ce code : il est calculé au moment de l'implémentation en se servant de *TABCORR* et du couple *(i,k)*.

c) ajouter en fin de procédure le code d'une action *"sortie-k"*.

La seule difficulté dans ce code est la détermination des ensembles *ENTREE* et *WAIT*.

**) code pour déterminer l'ensemble ENTREE.*

*Rappel de la définition de cet ensemble :*

$$ENTREE = \{\ 1 \leq j \leq p \text{ tq } - j \in list(k) \\ - count\text{-}mutex(j) > 0 \\ - CE(j) \text{ est vrai}\ \}$$

Nous utiliserons pour ce code deux variables temporaires, qui seront déclarées au début de l'action *"sortie-k"* : ce seront les variables *j* (de type entier) et *ENTREE* (de type "ensemble d'entiers")

```
"ENTREE := [ ] ;
 for j = 1 to p
 do  if   LIST[k,j] = 1            { j ∈ list(k) }
     then if   count-mutex[j] > 0
          then if   CE(j)
               then ENTREE := ENTREE+[j] ;"
```

**) Code pour déterminer l'ensemble WAIT.*

*Rappel de la définition de cet ensemble* (cfr V.24) :

$$"WAIT = \{\ IDij \text{ tq } - j \in list(k) \\ - count(IDij) > 0 \\ - CE(j) \text{ est vrai} \\ - (ai{:}B) \text{ est vrai}\ \}$$

c-à-d

$$WAIT = \{\ 1 \leq t \leq nbrwait \text{ tq } - TABCORR[t,2] \in list(k) \\ - count(t) > 0 \\ - CE(TABCORR[t,2]) \text{ est vrai} \\ (a_{TABCORR[t,1]}: B) \text{ est vrai}\ \}$$

Le code pour déterminer *WAIT* est alors le suivant :

```
"WAIT = [ ] ;
for t = 1 to nbrwait
do  if   LIST[k, TABCORR[t,2]] = 1
    then if   count[t] > 0
         then if   CE(TABCORR[t,2])
              then if   COND(TABCORR[t,1])
                   then WAIT := WAIT+[t] ;"
```

N.B. : + ce code utilise deux variables temporaires : ce sont les variables *t* (de type entier) et *WAIT* (de type "ensemble d'entiers").

+ la fonction **COND** est une fonction de type booléenne qui a été déclarée dans le monitor. Cette fonction reçoit une valeur entière $i$ ($1 \leq i \leq c$), et revoie la valeur TRUE ssi *(ai:B)* est vrai. (cfr. A.6)

*Nous pouvons maintenant donner le code d'une action "sortie-k" :*

N.B. : nous utiliserons dans ce code une procédure de nom ***selectionner***, que l'on supposera déclarée dans le monitor. L'appel de procédure **selectionner** *(E,W,x,e)* où $E$ et $W$ seront 2 ensembles d'entiers (l'un au moins doit être non vide !) aura pour effet de placer dans $x$ l'élément le plus "vieux" de $(E \cup W)$ et de placer dans $e$ la valeur $1 \Leftrightarrow x \in E$.

```
"P(action) ;
n[k] := n[k]-1 ;
begin var t,j,x,e : integer ;
          ENTREE,WAIT : set of integer ;

      ENTREE := [ ] ;
      for j = 1 to p
      do  if   LIST[k,j] = 1
          then if   count-mutex[j] > 0
               then if   CE(j)
                    then ENTREE := ENTREE+[j] ;
```

```
WAIT := [ ] ;
for t = 1 to nbrwait
do  if   LIST[k, TABCORR[t,2]] = 1
    then if   count[t] > 0
         then if CE(TABCORR[t,2])
              then if   COND(TABCORR[t,1])
                   then WAIT := WAIT+[t] ;

while (ENTREE ≠ [ ]) OR (WAIT ≠ [ ])
do    begin selectionner (ENTREE, WAIT, x, e) ;

            if   e = 1        { x ∈ ENTREE }
            then begin V(mutex[x]) ;
                       count-mutex[x] := count-mutex[x]-1 ;
                       n[x] := n[x]+1 ;
                       if   count-mutex[x] = 0
                       then ENTREE := ENTREE - [x] ;

                       for  j = 1 to p
                       do   if (j in ENTREE) and
                               (LIST[x,j] = 1)
                            then ENTREE := ENTREE - [j] ;

                       for  t = 1 to nbrwait
                       do   if (t in WAIT) and
                               (LIST[x,TABCORR[t,2]] = 1)
                            then WAIT := WAIT - [t]
                 end

            else begin V(s[x]) ;      { x ∈ WAIT }
                       count[x] := count[x]-1 ;
                       n[TABCORR[x,2]] := n[TABCORR[x,2]]+1 ;
                       if   count[x] = 0
                       then WAIT := WAIT - [x] ;

                       for  j = 1 to p
                       do   if (j in ENTREE) and
                               (LIST[TABCORR[x,2],j] = 1)
                            then   ENTREE   :=   ENTREE   - [j] ;
```

```
                                    for  t = 1 to nbrwait
                                    do     if (t in WAIT) and (LIST
                                             [TABCORR [x,2] ,TABCORR
                                             [t,2]] = 1)
                                           then WAIT := WAIT - [t]
                                end
                        end    {recommencer le cycle}
        end ;
        V(action) ;"
```

d) Pour chaque appel de procédure :

- ajouter juste . avant l'appel, le code d'une action *sortie-k*.
  . après l'appel, le code d'une action *entrée-k*.

A.II.3. EXEMPLE.

Supposons que l'on ait un monitor concurrent qui contienne 4 procédures *p1, p2, p3, p4,* avec les contraintes d'exclusion suivantes : $p1 \longleftrightarrow \{p1,p2\} \Rightarrow list(1) = \{1,2\}$
$p2 \longleftrightarrow \{p1,p3\} \Rightarrow list(2) = \{1,3\}$
$p3 \longleftrightarrow \{p2,p3\} \Rightarrow list(3) = \{2,3\}$
$p4 \longleftrightarrow \{\} \Rightarrow list(4) = \{\}$
(N.B. : $p = 4$)

De plus, le monitor contient 3 variables conditions *a1, a2, a3* ($\Rightarrow c = 3$) et des opérations *wait(a1)* utilisées dans P1, P2, P3
*wait(a2)* utilisées dans *P1, P2*
*wait(a3)* utilisées dans *P2*

Les différentes actions *wait(ai)j* possibles sont donc :

*wait(a1)1* $\longleftrightarrow$ 1 (identificateur)
*wait(a1)2* $\longleftrightarrow$ 2
*wait(a1)3* $\longleftrightarrow$ 3
*wait(a2)1* $\longleftrightarrow$ 4
*wait(a2)2* $\longleftrightarrow$ 5
*wait(a3)2* $\longleftrightarrow$ 6

(*nbrwait = 6*)

N.B. : - le tableau *LIST* devra donc avoir le contenu suivant :

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | **1** | **1** | x | x |
| 2 | **1** | x | **1** | x |
| 3 | x | **1** | **1** | x |
| 4 | x | x | x | x |

x désigne une valeur quelconque, différente de 1

- le tableau *TABCORR* devra avoir le contenu suivant :

| | 1 | 2 |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 1 | 2 |
| 3 | 1 | 3 |
| 4 | 2 | 1 |
| 5 | 2 | 2 |
| 6 | 3 | 2 |

1. *Modification des données du monitor*.

Les seules modifications sont la création de variables suivantes

```
"action      : semaphore ;
 mutex       : array [ 1..4] of semaphore ;
 s           :array [ 1..6 ] of semaphore ;
 count-mutex : array [ 1..4] of integer ;
 count       : array [ 1..6] of integer ;
 n           : array [ 1..4] of integer ;

 LIST      : array [ 1..4] of array [ 1..4] of integer ;
 TABCORR   : array [ 1..6] of array [ 1..2] of integer ;
```

2. *Modification du code d'initialisation du monitor*.

Le code suivant est ajouté dans le code d'initialisation :

```
"Init(action,1) ;
 begin var i : integer ;
       for i = 1 to 4
       do  begin Init(mutex[i],0) ;
                 count-mutex[i] := 0 ;
                 n[i] := 0
           end ;
       for i = 1 to 6
       do  begin Init(s[i],0) ;
                 count[i] := 0
           end
 end ;
 LIST := (1,1,0,0,1,0,1,0,0,1,1,0,0,0,0,0) ;
 TABCORR := (1,1,1,2,1,3,2,1,2,2,3,2) ; "
```

3. *Modification des procédures du monitor*.

Nous allons montrer le code des modifications à apporter dans une procédure *proc(1)*. Le principe sera identique pour les procédures *proc(2)*, *proc(3)* et *proc(4)*.

a) Nous aurons en-tête de la procédure et juste après chaque appel de procédure le code d'une action *entrée-1*, c-à-d le code décrit en A.15, où la lettre *k* est remplacée par *1*.

N.B : **CE**(1) renverra la valeur TRUE $\Leftrightarrow \forall\ 1 \leq i \leq 4 : LIST(1,i)=1 \Rightarrow n(i)=$
$\Leftrightarrow n(1) = 0$ et $n(2) = 0$

b) nous aurons en fin de la procédure et juste avant chaque appel de procédure le code d'une action *sortie-1*, c-à-d le code décrit en A.17-19, où la lettre *k* est remplacée par *1* ; la lettre *p* est remplacée par *4* et la lettre *nbrwait* par *6*.

c) une procédure *proc(1)* peut utiliser les opérations.*wait(a1)* qui seront codées comme des actions *wait(a1)1* (cfr A.15-16). *ID11* sera égal à *1*.

.*wait(a2)*

qui seront codées comme des actions *wait(a2)1*.
*ID21* aura comme valeur *4*.

# BIBLIOGRAPHIE

1. BRINCH HANSEN, P.
"The programming language concurrent Pascal."
IEEE Trans. on soft. Engin., vol. SE-1, pp. 199-206,
1975 (June)

2. DIJKSTRA, E.W.
"Cooperating sequential processes."
in "Programming Languages" (Ed. F.Genuys)
Academic Press, New York,
1968

3. BRINCH HANSEN, P.
"Distributed Processes : A concurrent programming concept."
CACM, vol. 21, pp. 934-941,
1978 (November)

4. MAO, T.W., and YEH, R.T.
"Communication port : A language concept for concurrent programming."
IEEE Trans. on Soft. Engin., vol. SE-6, pp. 194-204,
1980 (March)

5. ANDREWS, G.R.
"Synchronizing resources."
ACM Toplas, vol. 3, pp. 405-430,
1981 (October)

6. WIRTH, N.
"The programming language Pascal."
Acta Informatica, vol. 1, pp. 35-63,
1971

7. BAER, J.L.
"A survey of some theoretical aspects of multiprocessing."
ACM comput. surv., vol. 5, pp. 31-65,
1973 (March)

8. BRINCH HANSEN, P.
"Concurrent programming concepts."
ACM Comput. Surv., vol. 5, pp. 223-245,
1973 (December)

9. HOARE, C.A.R.
"Monitors : An operating system structuring concept."
CACM, vol. 17, pp. 549-557,
1974 (October)

10. HABERMANN, A.N.
"Synchronization of Communicating Processes."
CACM, vol. 15, pp. 171-176,
1972 (March)

11. BRINCH HANSEN, P.
"Structured multiprogramming."
CACM, vol. 15, pp. 574-578,
1972 (July)

12. ANDREWS, G.R., and SCHNEIDER, F.B.
"Concepts and Notations for Concurrent Programming."
ACM Comput. Surv., vol. 15, pp. 2-39,
1983 (March)

13. DIJKSTRA, E.W.
"The structure of the "THE" multiprogramming system."
CACM, vol. 11, pp. 341-346,
1968 (May)

14. VAN WIJNGAARDEN, A., MAILLOUX, B.J., PECK, J.L., KOSTER, C.H.A.
SINTZOFF, M., LINDSEY, C.H., MEERTENS, L.G.L.T., and FISKER,
R.G.
"Revised report on the algorithm language ALGOL68."
Acta INformatica, vol. 5, pp. 1-236,
1975

15. BRINCH HANSEN, P.
"A comparaison of two synchronizing concepts."
Acta Informatica, vol. 1, pp. 190-199,
1972

16. BRINCH HANSEN, P.
"Operating System Principles."
Prentice-Hall, Englewood Cliffs, N.J.,
1973

17. DAHL, O.J.
"Hierarchical program structures."
in "Structured Programming"
Academic Press, New York,
1972

18. HADDON, B.K.
"Nested monitor calls."
Oper. Syst. Rev., vol. 11, pp. 18-23,
1977 (October)

19. LISTER, A.
"The problem of nested monitor calls."
Oper. Syst. Rev., vol. 11, pp. 5-7,
1977 (July)

20. PARNAS, D.L.
"The non-problem of nested monitor calls."
Oper. Syst. Rev., vol. 12, pp. 12-14,
1978 (January)

21. WETTSTEIN, H.
"The problem of nested monitor calls revisited."
Oper. Syst. Rev., vol.12, pp.19-23,
1978 (January)

22. GUTTAG, J.
"Abstract data types and the development of data structures."
CACM, vol. 20, pp. 396-404,
1977 (June)

23. GUTTAG, J. and HORNING, J.
"The algebric specification of abstract data types."
Acta Informatica, vol. 10, pp.27-52,
1978

24. LISKOV, B. and ZILLES, S.
"An introduction to formal specification of data abstractions."
in "Current trends in programming methodology, vol. 1,
Software specification and design."
Prentice-Hall, Englewood Cliffs, N.J.,
1977

25. ZIMMERMAN, H.
"A standard layer model."
in "Computer Network Architecture and Protocols."
Plenum, New York (Ed. P.E.Green, Jr.)
1982

26. MITCHELL, J.G., MAYBURY, W., and SWEET, R.
"Mesa language manual, version 5.0."
Rep. CSL-79-3, XEROX P.A.R.C.
1979 (April)

27. WIRTH, N.
"Modula : a language for modular multiprogramming."
Softw. Pract. Exper., vol. 7, pp.3-35,
1977

28. WIRTH, N.
"Programming in Modula-2."
Springer-verlag, New-York,
1982

29. U.S. DEPARTMENT OF DEFENSE
"Programming Language Ada : Reference Manual."
in "Lecture Notes in Computer Science", vol. 106,
Springer-verlag, New-York
1981

30. COURTOIS, P.J., HEYMANS, F., PARNAS, D.L.
"Concurrent control with readers and writers."
CACM, vol. 14, pp. 667-668,
1971 (October)

31. KESSELS, J.L.W.
"An alternative to event queues for synchronisation in Monitors."
CACM, vol. 20, pp. 500-503,
1977 (July)